RIASSUNTI HIGH PERFORMANCE COMPUTING

CAPITOLO 1 – INFORMATION THEORY

La teoria dell'informazione è una teoria matematica relativa agli aspetti simbolici dell'informazione ed un approccio quantitativo al concetto d'informazione.

La teoria dell'informazione risponde alle seguenti domande:

- come immagazzinare e trasmettere l'informazione in un modo compatto? -> *compressione*
- Qual è il massimo ammontare d'informazione che può essere trasmessa su un canale? -> *capacità*
- Come si può proteggere la nostra informazione (da accessi non autorizzati o da errori di trasmissione)?

Il termine **informazione**, nel contesto della teoria dell'informazione, ha un significato preciso che è differente da quello che la nostra esperienza quotidiana fornisce. In una comunicazione l'informazione è scambiata attraverso dei messaggi.

La teoria dell'informazione è stata introdotta da Shannon nel 1948, la prima grande intuizione era di considerare l'informazione come un qualcosa che aiuta a rispondere a delle domande: alle domande a cui si può rispondere mediante sì o "no" è possibile utilizzare una singola quantità di due valori chiamata *bit*. Distinguiamo:

L'**Information bit**: è una misura di quanto impariamo dal risultato di un esperimento casuale, ad esempio il lancio di una moneta: senza lanciare una moneta non sappiamo quale sia il risultato, la nostra migliore ipotesi è indovinare a caso. Se qualcun altro impara il risultato di un lancio casuale di una moneta, possiamo chiedergli quale fosse il risultato, imparando un bit di informazione. Nel caso di domande più complesse da N possibili risultati si ottengono $\log_2 N$ bits di informazioni.

Ogni cifra di una stringa binaria può trasportare un po' di informazioni, tuttavia non sempre accade. Ad esempio, nel messaggio "0101010101" sono contenute poche informazioni perché è una semplice ripetizione del pattern "01". Un messaggio *prevedibile* è anche un messaggio **compressibile.** Comprimere un messaggio significa eliminare tutte le ridondanze, salvando solo le parti con significato, ovvero le informazioni.

DEF: la quantità d'informazione contenuta in un messaggio è la dimensione della rappresentazione più piccola del messaggio.

Alcune volte aggiungiamo della ridondanza ai messaggi per rilevare e correggere errori che sono introdotti durante la fase di trasmissione, un esempio è il codice ISBN dei libri: ogni codice ha 13 cifre, l'ultima è una cifra di controllo calcolata dalle 12 precedenti utilizzando il seguente algoritmo numerico:

Ogni cifra, da sinistra a destra, è moltiplicata per 1 o per 3 alternativamente. I risultati prodotti sono aggiunti in modulo 10 per ottenere un valore tra [0,9], quest'ultimo viene sottratto da 10 per ottenere un risultato [1,10], se il risultato è 10 allora è sostituito con 0.

La **ridondanza** è un meccanismo di sicurezza, garantisce che l'informazione sarà ricevuto anche se il messaggio è stato danneggiato durante la trasmissione. Tutti i linguaggi hanno un meccanismo di sicurezza simile, fatto di schemi strutture ed insiemi di regole per renderli ridondanti; spesso non siamo a conoscenza di queste regole ma i nostri cervelli le apprendono automaticamente e le usano per controlla la validità dei messaggi ricevuti.

Uno dei successi della teoria dell'informazione di Shannon è l'aver fornito una definizione formale di ridondanza ed aver specificato quanta informazione può essere trasportata in un messaggio. Viene riassunto nel teorema di *Shannon-Hartley* sulla **capacità** di un canale di comunicazione:

$$C = B\log(1 + \frac{\mathrm{S}}{N})$$

C è la capacità del canale (b/s) dopo aver applicato la correzione degli errori, B è la banda del canale (Hz) ed S/N è il rapporto segnale/rumore (Watt).

La **self-information** è una misura dell'informazione contenuta in un messaggio m. Sia p(m) = P{m out of M} la probabilità che il messaggio m sia scelto da tutti i messaggi nello spazio M, la **self information** è data dall'equazione:

$$I(m) = \log\left(\frac{1}{p(m)}\right) =. -logp(m)$$

Spesso la base del logaritmo è 2, e la self information è espressa in bits.

L'**entropia** è la quantità media di informazione prodotta da una fonte stocastica di dati, sia X una variabile discreta casuale con PMF p(x), l'entropia H(x) è data da:

$$H(X) = -\sum p(x)\log\ p(x)$$

Normalmente la base del logaritmo è base 2, in questo caso l'entropia è espressa in bits, di seguito alcune proprietà:

$H(X) >= 0$

$H_m(X) = \left(\log_m n\right)H_n(X)$

Sia X una variabile casuale di Bernoulli con:

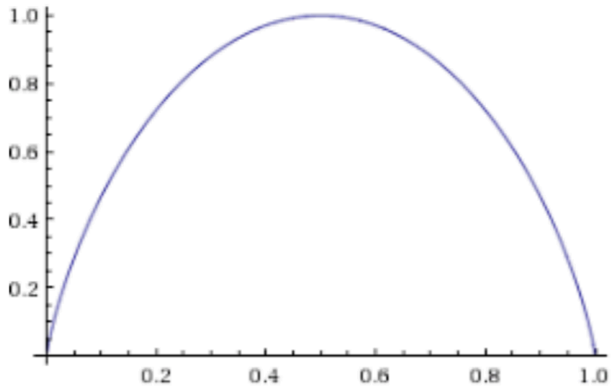
$P\{X = 1\} = p$

$P\{X = 0\} = 1 - p$

La funzione **entropia binaria** è data da:

$H_b\{p\} = p \log p - (1 - p)\log (1 - p)$

Prende un singolo numero reale come parametro. Mentre H(X) prende la PMF di una variabile casuale come parametro. L'entropia binaria è una funzione concava della distribuzione ed è uguale a 0 quando p=0 oppure 1. In questi casi la variabile non è casuale e non c'è incertezza, similmente il caso con più incertezza massima è quando p= ½ in cui $H_b(p) = 1$



La **joint entropy** prende in input due variabili discrete casuali con una distribuzione joint p(x,y)

$H(X,Y) = -\sum_x \sum_y p(x,y) \log p(x,y)$

La **conditional entropy** è il valore atteso delle entropie delle distribuzioni condizionali, mediato sulla variabile casuale di condizionamento.

$$H(Y|X) = \sum_x p(x)\, H(Y|X = x) = -\sum_x \sum_y p(x,y) \log p(y|x)$$

Chain rule = H (X, Y) = H(X) + H(Y|X)

Remark rule = H(X) – H(X|Y) = H(Y) – H(Y|X)

Considerando due PMFs p(x) e q(x), la funzione:

$$\mathrm{D(p||q)} = \sum_x p(x) \log \frac{p(x)}{q(x)}$$

È definita **entropia relativa** o **distanza Kullback-Leibler** tra le due PMFS. Può essere interpretata come il guadagno di informazioni raggiunto se q(x) è usata al posto di p(x).

E’ sempre non negativa, vale zero solo se p=q ma non è simmetrica e non soddisfa la disuguaglianza triangolare.

L’**informazione mutua** è la misura della dipendenza tra due variabili casuali:

$$I(X;Y) = \sum_x \sum_y p(x,y) \frac{\log p(x,y)}{p(x)p(y)}$$

È simmetrica in X ed Y ed è sempre non negativa, la definizione formale della **capacità di un canale** con input X ed output Y è $C = \max_{p(x)} I(X;Y)$

La **Kolmogorov complexity** misura l’informazione contenuta in una stringa x dalla lunghezza della più breve descrizione *d* di *x*. Tale lunghezza è denotata come *C(x)*. Possiamo pensare a *d* come una versione compressa di *x*, e l’algoritmo produce *x* come una procedura di decompressione.

Una *stringa x* è **Kolmogorov random** se $C(x) \geq |x|$, informalmente, è la proprietà di x di non essere più lungo di qualsiasi programma per computer in grado di produrre x. In altre parole, è la proprietà di x di essere incomprimibile.

AUTOMATA AND COMPUTABILITY

Gli **automi finiti** sono buoni modelli per i computer con una quantità di memoria estremamente limitata (ad esempio controlli per una porta automatica), possono essere *deterministici* o *non deterministici*.

Quando gli ***automi finiti deterministici*** **DFA** si trovano in uno stato determinato e leggono il prossimo segnale di input, sappiamo già quale sarà lo stato successivo. Formalmente è una tupla di 5 elementi (Q , $\sum$, δ, q_0 , F) dove:

1. Q è un insieme finito chiamato *stati*.
2. $\sum$ è un insieme finito chiamato *alfabeto*.
3. δ è la *funzione di transizione*.
4. q_0 è lo *stato iniziale*.
5. F è l’insieme degli *stati accettabili.*

Intuitivamente un DFA rappresenta un sistema che può essere in un insieme finito di stati differenti, come conseguenza di alcuni input (in un alfabeto finito), il DFA realizza una transizione da uno stato corrente ad un altro. L’output è accettato se, dopo aver letto l’ultimo input, il DFA entra nello stato accettato, altrimenti lo rifiuta. Il DFA può essere rappresentato per mezzo di una tabella chiamata *tabella di transizione di stato*. Una rappresentazione molto più suggestiva di un DFA è il *diagramma di stato*, in cui l’automa è rappresentato per mezzo di un grafo orientato, dove i nodi sono stati e gli archi sono transizioni, quest’ultimi sono etichettati con il simbolo la cui lettura causa la transizione stessa. Lo *stato iniziale* è un nodo con una freccia di immissione mentre gli stati di accettazione sono nodi a doppio cerchio.

Sia M = (Q , $\sum$, δ, q_0 , F) un DFA, se A è l'insieme di tutte le stringhe che M accetta, allora diciamo che A è un linguaggio della macchina M e scriveremo L(M) = A. Inoltre, diciamo che M riconosce A, un DFA può accettare diverse stringe, ma riconosce sempre solo un linguaggio. Un linguaggio è definito **linguaggio regolare** se alcuni DFA lo riconoscono.

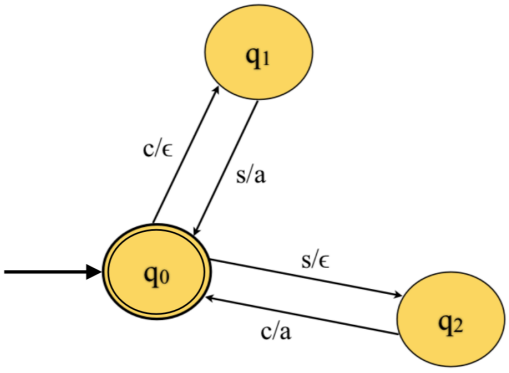
Un **finite state transducer (FST)** è un DFA i cui output sono stringhe (e non solo ''accettato'' o rifiutato''). Formalmente si tratta di una 7-tupla (Q , $\sum$, δ, q_0 , F, O, η) dove:

1. Q è un insieme finito chiamato *stati*.
2. $\sum$ è un insieme finito chiamato *alfabeto*.
3. δ: $Qx\sum \rightarrow$ Q è la *funzione di transizione*.
4. q_0 è lo *stato iniziale*.
5. F è l'insieme degli *stati accettabili.*
6. O è l'insieme finito di simboli output
7. η: $Qx\sum \rightarrow$ O è la funzione output

Un esempio di FST è un robot manufatturiero che riceve piattini e piccole tazze su un nastro trasportatore in input e posiziona su un nastro trasportatore di output le piccole tazze su un piattino. Alcune regole: se entrambe le mani sono libere, il robot afferra l'oggetto di input con la mano sinistra; se il robot ha già un oggetto nella mano sinistra, il nuovo oggetto deve essere diverso ed è afferrato con la mano destra. Il robot mette la piccola tazza sul piattino e mette l'elemento assemblato sul nastro trasportatore di uscita, quando la sequenza termina entrambe le mani del robot devono essere libere.

FST that models the robot:

c = small cup
s = saucer
a = assembled item



Il **non-determinismo** è una generalizzazione del determinismo, in una macchina non-deterministica diverse scelte possono esistere per lo stato successivo in ogni punto. Formalmente un automa finito non-deterministico **NFA** è una 5-tupla elementi (Q, $\sum$, δ, q_0 , F) dove:

1. Q è un insieme finito chiamato *stati*.
2. $\sum$ è un insieme finito chiamato *alfabeto*.
3. δ: $Q \times \sum_\varepsilon \rightarrow P(Q)$ è la *funzione di transizione*.
4. q_0 è lo *stato iniziale*.
5. F è l'insieme degli *stati accettabili.*

$\sum_\varepsilon = \sum \cup \{\varepsilon\}$ e P(Q) è l'insieme di potenza di Q, cioè la raccolta di tutti i sott'insiemi di Q. Il simbolo ε rappresenta la stringa vuota. L'NFA prende uno stato e un simbolo input o la stringa vuota e produce l'insieme dei possibili stati successivi.

Dopo aver letto un simbolo di input, se dallo stato attuale ci sono più modi per procedere, la macchina si divide in più copie di sé stessa e segue tutte le possibilità in parallelo. Durante il calcolo, alcune copie della macchina muoiono. Se una qualsiasi delle copie della macchina è in uno stato di accettazione alla fine dell'input, l'NFA accetta la stringa di input.

Sorprendentemente, DFAs ed NFAs riconoscono le stesse classi di linguaggi, un linguaggio è chiamato **linguaggio regolare** se alcuni NFA lo riconoscono. È utile perché descrivere un NFA per un determinato linguaggio alcune volte è molto più semplice rispetto a descrivere un DFA per lo stesso linguaggio.

Molti linguaggi non possono essere descritti dal significato degli automi finiti, gli **automi pushdown** sono simili agli automi finiti ma hanno una componente extra chiamata *stack*. La stack provvede memoria addizionale extra oltre la quantità finita disponibile nel controllo, consentendo agli automi pushdown di riconoscere alcune lingue non regolari. Così come per gli automi finiti, i pushdown possono essere *deterministici* o *non-deterministici*

I **deterministic pushdown automaton (DPA)** sono dei DFA arricchiti con una memoria ausiliaria strutturata come una stack. I simboli possono essere inseriti ed estratti dallo stack mediante la policy *LIFO (Last In First Out)*, l'ultimo simbolo inserito è il primo ad essere rimosso. Formalmente un DPA è una 6-tupla (Q, Γ , $\sum$, δ, q_0 , F) dove:

1. Q è l'insieme finito degli *stati*
2. $\sum$ è l'insieme finito chiamato *alfabeto*
3. Γ è l'insieme finito dell'*alfabeto stack*
4. δ: $Q \times \Gamma \times \sum \rightarrow Q \times \Gamma$ è la *funzione di transizione*
5. $q_0 \epsilon Q$ è lo *stato iniziale*
6. $F \subseteq Q$ è l'*insieme degli stati accettabili*

I **deterministic pushdown transducer (DPT)** è un DPA il cui output è una stringa e non soltanto un "accettato" o "rifiutato". È una 8-tupla (Q, Γ , $\sum$, δ, q_0, F, O, η):

1. Q è l'insieme finito degli *stati*
2. $\sum$ è l'insieme finito chiamato *alfabeto*
3. Γ è l'insieme finito chiamato *alfabeto stack*
4. δ: $Q \times \Gamma \times \sum \rightarrow Q \times \Gamma$ è una *funzione di transizione*
5. $q_0 \epsilon Q$ è lo *stato iniziale*
6. $F \subseteq Q$ è l'*insieme degli stati accettabili*
7. O è l'insieme finito dei *simboli output*
8. η: $Q \times \Gamma \times \sum \rightarrow O$ è la *funzione output*

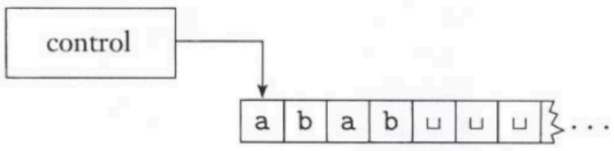
Esempio dell'automa visto in precedenza: in questo caso se il robot ha un oggetto in una delle due mani e riceve un altro oggetto dello stesso tipo mette il nuovo oggetto sulla cima dello stack. Se il robot ha un oggetto in una delle due mani e riceve un oggetto diverso, crea l'oggetto assemblato e lo mette sul nastro trasportatore. Se il robot ha le mani libere e riceve un oggetto e lo stack contiene l'oggetto dell'altro tipo allora il robot prende l'oggetto dalla cima dello stack e lo mette sul nastro trasportatore. Se lo stack avesse capacità infinita, nessun FST può modellare un robot con infiniti numeri di possibili stati.

Un **non-deterministic pushdown automaton (NPA)** è definito formalmente da una 6-tupla (Q, $\Gamma, \sum, \delta, q_0, \mathrm{F}$):

1. Q è l'insieme finito degli *stati*
2. $\sum$ è l'insieme finito chiamato *alfabeto*
3. Γ è l'insieme finito chiamato *alfabeto stack*
4. δ: $Q \times \Gamma_\varepsilon \times \sum_\varepsilon \rightarrow P(Q \times \Gamma_\varepsilon)$ è una *funzione di transizione*
5. $q_0 \epsilon\ Q$ è lo *stato iniziale*
6. $F \subseteq Q$ è l'*insieme degli stati accettabili*

Dove $\sum_\varepsilon = \sum \cup \{\varepsilon\}, \Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$ e $P(Q \times \Gamma_\varepsilon)$ è l'insieme di potenza di $Q \times \Gamma_\varepsilon$, ovvero la collezione di tutti i sott'insiemi di $Q \times \Gamma_\varepsilon$. ε rappresenta la stringa vuota.

Gli automata pushdown sono limitati a causa della politica rigida LIFO che è usata per controllare la memoria stack, una **Macchina di Turing** è un modello molto più accurato per un computer general purpose. Il modello della macchina di Turing usa un nastro infinito come memoria illimitata. Il nastro (tape) ha una testa che può leggere e scrivere simboli e muoversi sul nastro. Inizialmente il nastro contiene solo le stringhe di input ed è vuoto nelle parti restanti. Per salvare l'informazione la macchina scrive sul nastro, per leggere le informazioni la macchina muove la testa del nastro in avanti e indietro sulle celle del nastro che contengono le informazioni. In qualche punto la macchina può fermare il calcolo e restituire come output "accettato" o "rifiutato". Se la macchina non entra in nessuno dei due stati allora prosegue all'infinito, senza mai fermarsi.



La collezione di stringhe che una macchina di Turing M accetta è il linguaggio di M, o il linguaggio riconosciuto da M, denotato come L(M). Un linguaggio è **riconoscibile** se qualche macchina di Turing lo riconosce. Un linguaggio è **decidibile** se qualche macchina di Turing "lo decide", ovvero se restituisce sempre la decisione di accettare o rifiutare.

Formalmente, la macchina di Turing è una 7-tupla (Q, $\Gamma, \sum, \delta, q_0, q_{accept}, q_{reject}$)

1. Q è l'insieme finito degli *stati*

2. $\sum$ è l'insieme finito chiamato *alfabeto*
3. Γ è l'insieme finito chiamato *alfabeto stack*
4. δ: $Q \times \Gamma \rightarrow \ Q \times \Gamma \ \times \{L, R\}$ è la *funzione di transizione*
5. $q_0 \epsilon \ Q$ è lo *stato iniziale*
6. $q_{accept} \in Q$ è lo stato accettato
7. $q_{reject} \in Q$ è lo stato rifiutato

Alcuni varianti sono la multi-tape TM, nondeterministic TM e la probabilistic TM.

Per una **Macchina di Turing K-Tape** la *funzione di transizione* è: δ: $Q \times \Gamma^k \rightarrow \ Q \times \Gamma^k \ \times \{L, R, S\}^k$

Una *configurazione* di una k-tape TM è formata dallo stato del dispositivo di controllo, dal contenuto dei k-nastri e dalla posizione delle k-teste.

La configurazione iniziale:

- Il dispositivo di controllo è nello stato iniziale
- Il primo nastro (nastro di input) contiene un numero finito di simboli non vuoti
- I nastri di memoria contengono un simbolo speciale Z_0 nella loro prima cella e nessun altro simbolo non vuoto.
- L'ultimo nastro (tape di uscita) è vuoto
- Le teste sono posizionate sulla prima cella dei loro nastri

Esempio di TM: robot che mette insieme una sequenza di grandi piatti I, piattini s e piccole tazze c. Le regole sono:

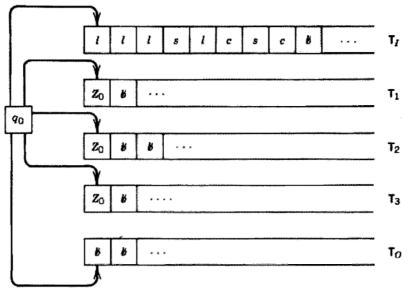
1. Ricevere tutti gli oggetti dal nastro e mettere gli oggetti sulla cima di 3 pile differenti
2. Dopo aver ricevuto tutti gli oggetti, estrarre un I, un s ed un c da ognuno degli stack e produrre l'oggetto assemblato a.
3. Alla fine delle operazioni, controllare se tutti gli stack sono vuoti.

Il controllore della TM che modella questo sistema ha 5 stati:

- q_0, stato iniziale
- q_R, stato della lettura dell'input
- $q_{W,}$ stato della scrittura dell'output
- q_A, stato finale "accettato"
- q_E, stato finale errore

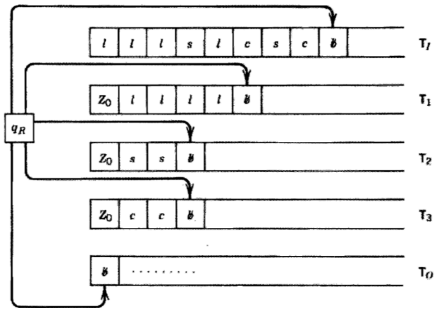
La macchina di Turing ha un nastro input T_I, tre nastri di memoria $\{ T_1, \ T_2, \ T_3\}$ ed un nastro output T_O.

1) Start from q_0 and switch to q_R, letting unchanged the symbols Z_0 on the memory tapes and moving the corresponding heads to the right. Do not read anything from T_I and do not move the heads of T_I and T_O.

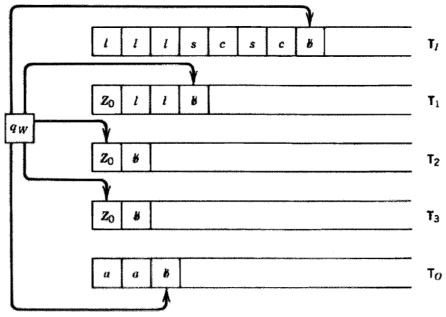


2) When arriving in q_R and an *l*, *s* or *c* is read from tape T_I, stay in q_R and write input symbols on the suitable memory tape. Move the heads of T_I and of memory tapes to the right, and do not move the head of T_O.

3) When arriving in q_R and an empty symbol is read from T_I, it does mean that all input objects have been collected. Do not move the head of T_I anymore, switch to q_W, move to the left the heads of the memory tapes, and do not write anything on T_O.



4) When arriving in q_W and symbols *l*, *s* e *c* are read from memory tapes, stay in q_W. Write an empty symbol in place of the read symbol and move the heads to the left. Moreover, write an *a* on T_O and move its head to the right.



5) When arriving in q_W and Z_0 is read from all memory tapes, it means that the same number of *l*, *s* and *c* has been received and that they have been assembled. Then switch to state q_A, write an empty symbol in place of Z_0 on the memory tapes, write symbol *Y* (success) on T_O, and halt.
In any other case, enter error state q_E, write symbol *N* (failure) on T_O, and halt.

Un **algoritmo** (effective computation) è una computazione che produce un risultato desiderato in un numero finito di passi, *Church-Turing thesis:* ogni effective computation può essere effettuato da una TM.

Una funzione f è una **funzione computabile** se alcune macchine di Turing M su tutti gli input w, terminano con f(w) sul loro nastro di output.

Certi problemi possono essere risolti mediante algoritmi e altri no, studiare la non risolvibilità è utile per realizzare che un problema deve essere semplificato o alterato prima di trovare un algoritmo risolutivo. Possiamo formulare problemi computazionali in termini di test dell'appartenenza ad una lingua, mostrare che il linguaggio è decidibile è lo stesso che mostrare che il problema computazionale è algoritmicamente risolvibile.

Possiamo determinare se un DFA accetta o meno delle stringhe:

$$E_{DFA} = \{< A > \mid A \text{ è } un\ DFA\ e\ L(A) = \emptyset\}$$

Teorema: E_{DFA} è un linguaggio decidibile

Inoltre, possiamo determinare se due DFAs riconoscono lo stesso linguaggio:

$$EQ_{DFA} = \{< A, B > \mid A\ e\ B\ sono\ DFAs\ e\ L(A) = L(B)\}$$

Teorema: EQ_{DFA} è un linguaggio decidibile

Ci sono numerosi problemi non risolvibili algoritmicamente, un esempio è il problema di determinare se una macchina di Turing accetta una stringa in input.

$$A_{TM} = \{< M, w > \mid M \text{ è } una\ TM\ e\ M\ accetta\ w\}$$

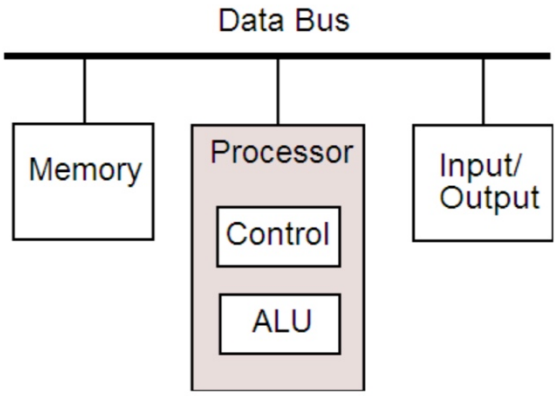
Teorema: A_{TM} è indecidibile.

A_{TM} però è *riconoscibile* dalla seguente TM:

U= "Sull'input <M,w>, dove M è una TM e w è una stringa: 1 simula M sull'input w, se M entra in uno stato "accept" allora accetta se M entra nello stato reject allora reject".

Se <M,w> è in A_{TM} allora U lo accetta, quindi U riconosce A_{TM}. Bisogna notare che U ripete in input <M,w> se M ripete w, che è il perché la macchina non decide A_{TM}. U è la **macchina universale di Turing**, una TM che è capace di simulare l'evoluzione di qualsiasi altra TM dalla ssua descrizion. La **UTM** ha giocato un ruolo importante nello sviluppo dei *stored-program computers*.

L'**architettura Von Neumann** è il modello principale per il design dei stored-program computers. E' caratterizzato da un'unità centrale di processamento (CPU) e una struttura di memoria che immagazzina sia i dati che i programmi. Le istruzioni sono eseguite una dopo l'altra, il primo computer elettronico basato su questa architettura era il EDVAC (1949).



Lo schema in figura rappresenta 5 componenti principali del modello:

- **CPU** o processore, che include un'unità logica aritmetica (ALU) e un'unità di controllo.
- **MEMORIA**, contiene i dati da essere processati e le istruzioni da eseguire (ovvero i programmi).
- **UNITA' INPUT**, attraverso cui i dati sono inseriti nel computer per essere processati
- **UNIT' OUTPUT,** necessaria per restituire i dati processati all'utente
- **DATA BUS,** un canale che connette tutte le componenti

La **indecidibilità** di A_{TM} può essere usata come prova dell'indecidibilità del **problema d'arresto (halting problem)**:

$$HALT_{TM} = \{M, W \mid M \text{ è } una\ TM\ ed\ M\ si\ arresta\ su\ un\ input\ w\}$$

Teorema dell'arresto di Turing: $HALT_{TM}$ è indecidibile.

DIMOSTRAZIONE: assumiamo che R è una TM che decide $HALT_{TM}$, costruiamo S, una TM per decidere A_{TM}:

S = "Sull'input <M,w>, dove M è una TM e w una stringa:

1. Esegui R sull'input <M,w>
2. Se R rifiuta -> rifiuta
3. Se R accetta, simula M su w fino a quando non si ferma
4. Se M accetta -> accetta, se M rifiuta -> rifiuta"

Chiaramente se R decide $HALT_{TM}$ allor S decide A_{TM}, che è impossibile c.v.d.

Un altro modo equivalente per descrivere il problema d'arresto è il seguente:

"Non esiste un programma P in grado di predire se un altro programma Q si fermerà dopo un numero finito di passaggi oppure no".

DIMOSTRAZIONE: supponiamo per assurdo che esiste un programma P in grado di fare quanto detto, allora possiamo modificare P per produrre un nuovo programma P' che, dato un altro programma Q come input, è in grado di:

1 eseguire per sempre se Q si arresta

2 fermarsi se Q viene eseguito per sempre

In questo modo, se utilizziamo come Q per P' il proprio codice, allora P' verrà eseguito per sempre se si ferma oppure si ferma se viene eseguito per sempre -> ASSURDO c.v.d.

Il problema d'arresto ci dice che il problema generale di verifica software è algoritmicamente irrisolvibile, una *program specification* spesso include un'istruzione secondo cui il programma dovrebbe produrre una risposta di qualche tipo, ma controllare se il programma alla fine produce un risultato è equivalente al problema dell'arresto che è indecidibile.

Il metodo primario per provare che i problemi sono computazionalmente irrisolvibili è chiamato **riducibilità**, una riduzione è un modo per convertire un problema in un altro in modo da avere una soluzione al secondo problema che può essere usato per risolvere il primo.

La riducibilità includerà sempre due problemi, che possiamo chiamare A e B:

Se A è *riducibile* a B allora B è *decidibile*, quindi anche A è *decidibile*. Equivalentemente se A non è *decidibile* e *riducibile* a B allora B non è *decidibile.*

Spesso mostriamo che un problema è indecidibile dimostrando che il problema d'arresto riduce quel problema. Se A è riducibile a B e B è riducibile ad A, allora diciamo che A e B sono problemi equivalenti.

Ci sono diversi modi per definire la nozione di riduzione di un problema ad un altro, un tipo semplice di riducibilità è chiamata **mapping reducibility**, come al solito rappresentiamo i problemi computazionali come linguaggi:

Il linguaggio A è **mapping riducibile** ad un linguaggio B, scritto con $A \leq B_m$, se c'è una funzione computabile f, dove per ogni w:

$$w \in A \leftrightarrow f(w) \in B$$

La funzione f è chiamata riduzione di A a B.

La **riducibilità Turing** è una generalizzazione della mapping reducibility. Un **oracolo** per la lingua B è un dispositivo esterno in grado di decidere un linguaggio B; quindi, di stabilire se ogni stringa w è un membro di B. Un **oracle Turing Machine** è una TM modificata che ha la capacità addizionale di interrogare un oracle ed ottenere le risposte in un singolo step computazionale. Scriviamo M^B per descrivere un oracle Turing machine che ha un oracolo per il linguaggio B.

Il linguaggio A è **Turing reducible** al linguaggio B, $A \leq B_T$, se A è **decidibile relativo per B**, che significa che esiste un'oracle Turing Machine che è in grado di decidere A.

Teorema: se $A \leq B_T$ e B è decidibile, allora anche A è decidibile.

DIMOSTRAZIONE: se B è decidibile, allora possiamo sostituire l'oracle per B con una procedura che decide B. Quindi, possiamo sostituire l'oracle Turing Machine che decide A con una TM ordinaria che decide A. La **Turing reducibility** è una generalizzazione della mapping reducibility, ovvero è possibile utilizzare la mapping reduction per un oracle Turing machine che decide A relativamente a B ovvero: $A \leq B_m \rightarrow A \leq B_T$ (non il contrario)

ESEMPIO: Consideriamo i linguaggi A_{TM} *e* $\overline{A_{TM}}$, intuitivamente essi sono riducibili l'uno con l'altro. Invece essi sono Turing riducibili -> $\overline{A_{TM}}$ non è mapping riducibile per A_{TM} perché A_{TM} è Turing-riconoscibile ma $\overline{A_{TM}}$ non lo è.

COMPUTATIONAL COMPLEXITY

La **complessità computazionale**, o semplicemente complessità di un *algoritmo* è l'ammontare di risorse richieste per farlo eseguire. La complessità computazionale di un *problema* è la complessità minima di tutti i possibili algoritmi per quel problema. Una **classe di complessità** può essere pensata come una collezione di problemi computazionali che richiedono O(f(n)) risorse di una macchina astratta per essere risolti, dove n è la dimensione dell'input.

Diciamo che $\boldsymbol{g(n) = O(f(n))}$ se esistono delle costanti *a, b* tali che:

$g(n) < a(fn) + b \; \forall n$

Diciamo che $\boldsymbol{g(n) = \Omega(f(n))}$ se esistono delle costanti *a>0, b* tali che:

$g(n) > a(fn) - b \; \forall n$

Diciamo che $\boldsymbol{g(n) = \Theta(f(n))}$ se esistono delle costanti *a, b* tali che:

$g(n) = O(fn) \; e \; g(n) = \Omega(f(n)) \; \forall n$

Un **problema decisionale** è una domanda del tipo SI-NO sull'input dei valori. Risolvere un problema decisionale è equivalente a decidere il linguaggio *L={x:f(x)=1}*, dove f() è la funzione che definisce il problema.

R è la classe di tutti i linguaggi decidibili su qualche macchina di Turing.

Tutti i problemi che appartengono a R sono definiti da funzioni computazionali f, il problema d'arresto non è in R (come molti problemi decisionali).

Per molti problemi non sappiamo la loro complessità, ma possiamo porre dei limiti su di essa. Sappiamo che la complessità di un problema è in mezzo ad un upper bound ed un lower bound, scoprire migliori algoritmi riduce l'upper bound del caso peggiore della complessità computazionale.

Dimostrare le affermazioni sui limiti inferiori può essere difficile: dobbiamo dimostrare qualcosa che valga per tutti i possibili algoritmi che risolvono il problema (non solo gli algoritmi esistenti, ma quelli che potrebbero essere scoperti in futuro). Ogni dimostrazione migliore porta il limite inferiore verso l'alto.

La **time complexity TIME(f(n))** è la classe di linguaggi che sono decidibili da una macchina di Turing time-deterministic O(f(n)).

La **space complexity SPACE(f(n))** è la classe di linguaggi che sono decidibili da una macchina di Turing space-deterministic O(f(n)).

Un **algoritmo time-efficient** viene eseguito in tempo polinomiale $O(n^k)$

Un algoritmo che richiede un tempo super-polinomiale (esponenziale) per risolvere un problema non è time-efficient.

La scelta di un modello computazionale può incidere il tempo di complessità del linguaggio.

Teorema – Sia f(n) una funzione, dove $f(n) \geq n$. Quindi ogni macchina di Turing multitape f(n) tempo ha una macchina di Turing a nastro singolo a tempo O(f2(n)) equivalente.

Teorema - Sia f(n) una funzione, dove f(n) ≥ n. Quindi ogni macchina di Turing a nastro singolo non deterministico f(n) tempo ha una macchina di Turing a nastro singolo deterministica 2O(f(n)) tempo.

P è la classe di linguaggi che può essere decisa velocemente (tempo polinomiale) su un computer classico, ovvero: l'unione, su tutti gli interi positivi k, di TIME(n^k). In altre parole, la classe dei linguaggi che sono decidibili in tempi polinomiali da una macchina di Turing deterministica.

Esempi: ordinare gli elementi di una lista in un determinato ordine, trovare il massimo comune divisore di due interi.

EXP è l'unione, su tutti gli interi positivi, di TIME(2^{n^k}).

P/f(n) è la classe dei linguaggi che sono decidibili da una macchina di Turing tempo-deterministica, dato un input w di lunghezza n ed una stringa di avviso c di lunghezza f(n). La stringa di avviso è un extra input. **P/poly** -> lunghezza di c è polinomiale in n (importante per diverse ragioni, ad esempio nella criptografia la sicurezza è definita spesso contro avversari P/poly).

NC consiste in tutti i linguaggi che possono essere decisi in tempo poli-logaritmico su un parallel computer con un numero polinomiale di processori, più precisamente: trovare costanti k ed m tali che il linguaggio è deciso in tempo $O(\log_m n)$ usando $O(n^k)$ processori. **NC è un sott'insieme di P** (esempi di linguaggi NC sono: l'addizione, moltiplicazioni di interi, moltiplicazione di matrici, inversa di matrice, calcolo di rango e determinante di una matrice etc.)

Un **verifier** per un linguaggio A è un algoritmo V, dove: A = {w| V accetta {w,c} per alcune stringhe c}. Misuriamo il tempo di un verifier soltanto in termini di lunghezza di w.

NP è la classe dei linguaggi che hanno verifiers in tempo polinomiali, questo significa che se qualcuno da un'istanza di un problema ed un certificato (ovvero una prova) che la risposta sia SI, possiamo controllare che sia corretta in tempo polinomiali.

NTIME(t(n)) è la classe dei linguaggi che sono decidibili da una macchina di Turing non deterministica O(t(n)). NP è l'unione, su tutti gli interi positivi k, di NTIME(n^k)

coNP contiene i linguaggi che sono complementari dei linguaggi NP. Questo significa che se qualcuno ci da un'istanza del problema ed il certificato che la risposta sia NO allora possiamo controllare che il risultato sia corretto. Non sappiamo se coNP è una classe differente da NP, un esempio ne è il problema della fattorizzazione intera.

La **fattorizzazione intera** è descritta dal seguente problema: dato un numero intero N ed un intero M con $1 \leq M \leq N$, N ha un fattore primo d con $1 \leq d \leq M$?

È in NP, perché il fattore primo d, con $1 \leq d \leq M$, per N serve da "testimone" di un'istanza YES. Possiamo controllare il testimone in tempo polinomiale eseguendo la divisione N/d.

È in coNP, perchè una fattorizzazione prima di N senza fattori < M serve come "testimone" di un'istanza NO.

L'algoritmo classico migliore per la fattorizzazione di numeri interi è il *number field sieve*, la cui complessità temporale è super-polynomial in n = log N.

La **riduzione in tempo polinomiale (polynomial-time reducibility)** permette di classificare i problemi in base alle difficoltà *relative*.

Il linguaggio A è riducibile polynomial-time ad un linguaggio B, scritto con $A \leq B_p$, se esiste la funzione computazionale tempo-polinomiale f, tale che per ogni w:

$$w \in A \leftrightarrow f(w) \in B$$

La funzione f è chiamata **polynomial-time reduction** di A a B.

Se $A \leq B_p$ allora A non può essere più difficile di B, perché ogni volta che esiste un algoritmo efficiente per B allora ne esiste uno per A. Contrariamente, se non esiste un algoritmo per A allora non esiste neanche per B. Se $A \leq B_p$ e $B \leq A_p$ allora A e B sono **computazionalmente equivalenti**

Un problema B è definito **C-hard** se per tutti i problemi A in C si ha che $A \leq B_p$, se inoltre abbiamo che B è in C, allora B è chiamato **C-complete**. In generale **C** e **C-complete** sono classi differenti, ma non possiamo dire che la loro intersezione è vuota per tutto C. I problemi **C-complete** sono i problemi più difficili della classe C: se riusciamo a risolvere uno di loro in maniera efficiente, abbiamo automaticamente risolto tutti gli altri problemi in C. Invece se risolviamo in maniera efficiente un problema in C che non è C-Complete, non siamo in grado di risolvere efficientemente i problemi che sono in C-Complete. I problemi **C-hard** sono almeno difficili come i problemi più difficili di C (ovvero, i problemi C-complete).

I **problemi trattabili** sono risolti in tempo polinomiale, quindi sono in P. I problemi intrattabili richiedono tempo esponenziale. Se $A \leq B_p$ e B è trattabile allora anche A lo è, risolvendo prima B in tempo polinomiale ed utilizzando la polynomial-time reduction per risolvere A. Se $A \leq B_p$ ed A è intrattabile allora anche B lo è.

Il linguaggio B è **NP-hard** se ogni linguaggio A in **NP** è riducibile in tempo polinomiale a B. Quindi, i problemi **NP-hard** sono almeno difficili quanto ogni altro problema in NP. Il problema d'arresto non è NP, perché non è decidibile in un numero finito di operazioni ma è NP-HARD.

Il linguaggio B è **NP-Complete** se soddisfa due condizioni:

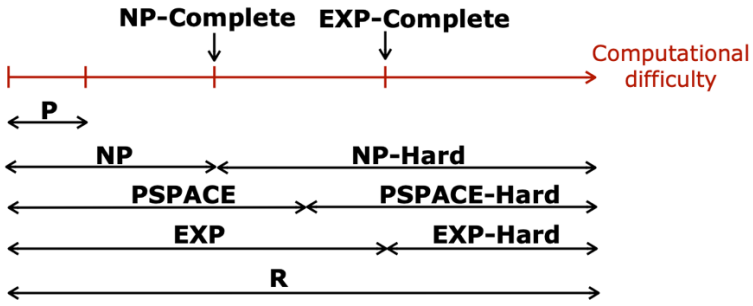
1. B è NP
2. B è NP-HARD

Tutti i problemi NP-Complete sono in un certo senso "lo stesso problema", poiché qualsiasi problema NP-Completo è riducibile in un tempo polinomiale in un altro problema NP-Completo in tempo polinomiale. (Esempi di problemi NP-Completi sono SAT, 3SAT, CircuitSAT, Map Coloring, Bin Packing etc.)

PSPACE è la classe dei linguaggi che sono decidibili in uno spazio polinomiale su una macchina di Turing deterministica. In altre parole, PSPACE è l'unione su tutti gli interi positivi k di SPACE(n^k).

⊆EXP: il numero totale di configurazioni della TM è al massimo 2q(n) per alcuni polinomi q(n)

⊇NP: scorrere tutti i certificati, memorizzandoli uno per uno nel nastro di lavoro della TM (assumendo una TM a tre nastri)



Chiaramente **P** è un *sott'insieme* di **NP** (se il problema può essere risolto in tempo polinomiale, può anche essere verificato in tempo polinomiale su una macchina di Turing deterministica). Comunque, non è così tanto chiaro se ci sono problemi in NP che non sono in P. Il problema P vs NP è uno dei problemi più importanti nella computer science teorica.

Se uno dei problemi NP-Hard si può risolvere velocemente, allora tutti i problemi NP possono essere risolti velocemente (che implicherebbe P=NP). Ci occuperemo dei problemi NP-Complete per via degli interessi pratici, e si cercherà di ottenere una soluzione per ogni loro problema. Dall'altra parte, provare che uno dei problemi NP-COMPLETE non abbia una soluzione efficiente significa che P≠NP.

Un **oracle** è una black box in grado di risolvere un problema decisionale in tempo O(1) (costante).

P^0 è la classe dei linguaggi che sono decidibili in tempo polinomiale, dato l'oracle access al problema O.

Example:
Let $\overline{SAT}$ denote the language of unsatisfiable formulae.
Then $\overline{SAT}$ belongs to $\mathbf{P}^{SAT}$.

PH è l'unione di tutte le classi della gerarchia polinomiale composta da:
$\mathbf{\Delta_0^P = \Sigma_0^P = \Pi_0^P = P}$ and, for i>0:

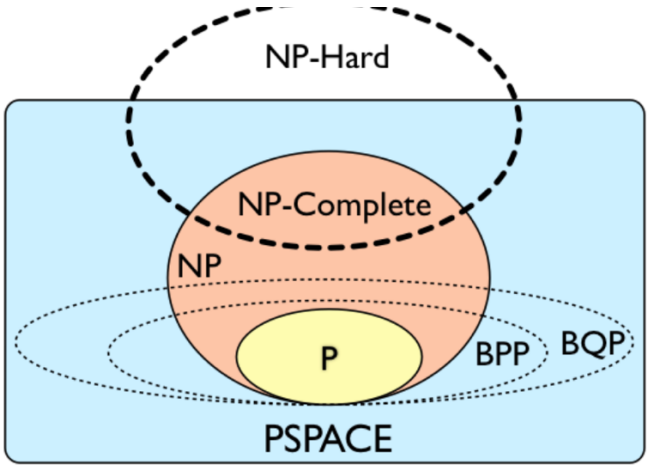
- $\mathbf{\Delta_i^P = P}$ with $\mathbf{\Sigma_{i-1}^P}$ oracle
- $\mathbf{\Sigma_i^P = NP}$ with $\mathbf{\Sigma_{i-1}^P}$ oracle
- $\mathbf{\Pi_i^P}$ = **coNP** with $\mathbf{\Sigma_{i-1}^P}$ oracle

#P è l'insieme di counting problem associati con i problemi decisionali in NP. Un problema in **#P** deve essere almeno difficile al corrispondente problema NP. Una TM tempo deterministica con un oracle **#P** può risolvere tutti i problemi in **PH**.

Nella metà degli anni 70', i **randomized algorithms** furono introdotti, suggerendo che alcuni problemi che non hanno una soluzione efficiente su una TM deterministica, possono avere una soluzione efficiente su una **TM probabilistica**, che è un tipo particolare di TM nondeterministica (ogni passaggio ha due possibili mosse e la probabilità di un branch è di 2^{-k} dove k è il numero di passi che presenti su quel branch. Il progetto generale di prendere algoritmi randomizzati e convertirli in algoritmi deterministici è chiamato **derandomizzazione**.

BPP (**B**ounded-error **P**robabilistic **P**olinymoial-time) è la classe dei linguaggi che sono decisi in tempo polinomalie da una TM probabilistica, con una probabilità di errore di 1/3. La relazione tra BPP ed NP è sconosciuta, non sabbiamo se BPP è un sott'insieme di NP o il viceversa. Invece c'è un'evidenza riguardo P = BPP.

BQP (**B**ounded-error **Q**uadratic **P**olinymoial-time) è la classe dei linguaggi che sono decisi in tempo polinomiale da un *universal quantum computer*, con un errore di probabilità pari ad 1/3. Si sa che BQP include BPP, ma dove si relazioni BQP rispetto a P, NP ed PSPACE è ancora sconosciuto. Cosa sappiamo è che i computer quantici possono risolvere tutti i problemi in P efficientemente, ma non ci sono problemi al di fuori di PSPACE che possono risolvere efficientemente. Per questo motivo, BPQ si trova in qualche posto tra P e PSPACE.



ARCHITETTURE PARALLELE

Tradizionalmente, il software è scritto per computer sequenziali, e quindi, per essere eseguito su macchine provviste di un'unica CPU. Un programma sequenziale è una sequenza di istruzioni che sono eseguite una dopo l'altra. Fino al 2000 le performance delle CPU crescevano rapidamente, principalmente per due ragioni: l'aumento della **clock speed** delle CPU (ovvero il numero di cicli di clock al secondo) e per l'utilizzo di **meno cicli di clock per operazione**. La performance di un processore sequenziale dipende dal **data transmission rate** dentro l'hardware. I *limiti assoluti* sono: la velocità della luce e la velocità di transmissione delle linee di rame. Un altro problema importante è il consumo di energia (Legge di Moore anche qui !). Negli anni 2000 nonostante il numero di transistori aumentava avvenne uno stop improvviso della clock speed (2-3 GHz). L'*instruction latency* (tempo per performare un operazione dall'inizio alla fine) non migliorò di molto e in alcuni casi la latenza peggiorava. Tutti i computer moderni hanno processori paralleli, la CPU di un desktop o un computer laptop gestisce facilmente centinaia di operazioni matematiche conteporanemente, inoltre sono caratterizzati da *CPU cores* multipli, ogni core ha *unità d'esecuzioni* che possono essere usate contemporaneamente. Ogni unità d'esecuzione può eseguire larghe operazioni vettoriali e l'esecuzione delle unità sono pipelined, quindi una non ha bisogno di aspettare la fine dell'istruzione precedente prima di cominciare la prossima.

La performance delle CPU odierne migliorar non nella riduzione della latenza ma nel **throughput** (numero di operazioni completate in una time unit) e per beneficiare da questo nuovo miglioramento bisogna utilizzare nuovi tipi di software che mettono in conto la differenza tra throughput e latenza. Il *parallelismo* è principalmente utilizzato nel contesto dell'**high performance computing** per applicazioni scientifiche ed industriali (analisi meteo, geologiche, ML & Big Data etc.), in questo contesto è interessante citare la strategia "oltre Moore" (more than moore strategy): piuttosto che migliorare i chip e lasciare che le applicazioni seguano (i chip), si inizierà dalle applicazioni e si lavorerà verso il basso per vedere quali chip sono necessari per supportarle".

I top 500 supercomputers sono classificati in base alle loro performance sul LINPACK benchmarck (il problema consiste nel risolvere un sistema denso di equazioni lineari). La versione adottata dal benchmarck permette all'utente di scalare la dimensione del problema ed ottimizzare il software in modo di ottenere la miglior performance per una certa macchina. Il Super Computer *Leonardo (*Bologna) è attualmente 4° nella classifica.

Nel 1966 Flynn propose una delle prime classificazioni per computer paralleli, nota come **Flynn's taxonomy**, che separa le architetture basate sulla realizzazione degli istruzioni e dei data flows. Questi flussi possono essere singoli o multipli

Flynn's taxonomy

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

SISD: **S**ingle **I**nstruction, **S**ingle **D**ata. Sono caratterizzati da un processore sequenziale e il flow delle istruzioni (costituito da una singola istruzione) è presentato alla CPU e viene processato. Viene utilizzato un singolo dato come input in un clock cycle, l'esecuzione è *deterministica* -> è l'architettura più tradizionale (PCs, workstations, mainframes etc)

Una **superscalar CPU architecture** implementa una forma di parallelismo chiamata **instruction-level parallelism** all'interno di un singolo processore. Consente quindi un throughput della CPU più veloce di quanto sarebbe altrimenti possibile alla stessa frequenza di clock, un processore superscalare esegue più di un istruzione durante il clock cycle inviando contemporaneamente più istruzioni alle unità funzionali ridondanti della CPU. Ogni unità funzionale è una risorsa esecutiva all'interno di una singola CPU come un'unità logica aritmetica, un bit shifter o un moltiplicatore. Di solito, CPU superscalari sfruttano il pipeling, che è un'altra tecnica d'incremento della performance.

SIMD: **S**ingle **I**nstruction, **M**ultiple **D**ata (**data level parallelism**). Alla CPU viene presentata una singola istruzione, ogni processore può operare su un dato differente, l'esecuzione è *sicnrona (deterministica)*. Questo tipo di macchine tipicamente ha: un distributore d'istruzioni, una larga banda nella connection network ed un grande numero di not-so-powerful processors. (Esempi array processor MP-2, Vector Supercomputer come l'IBM 9000).

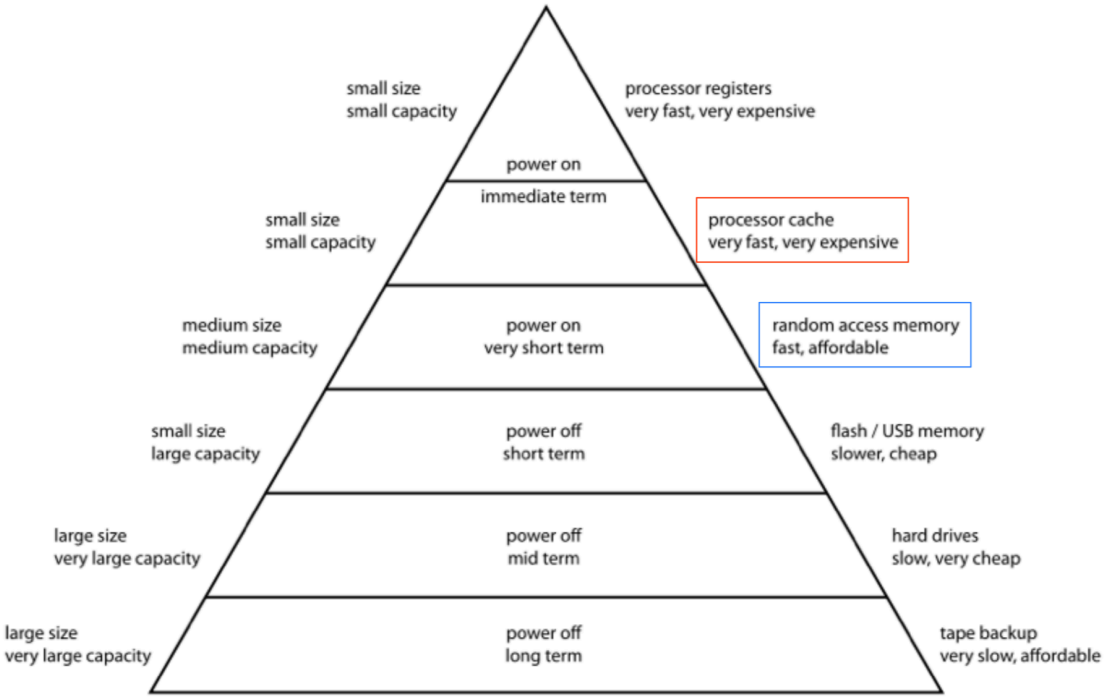
MISD: **M**ultiple **I**nstruction, **S**ingle **D**ata. E' caratterizzato da un unico data flow che nutre diverse CPU, ogni CPU processa il data flow indipendentemente dalle altre (architettura poco usata, un esempio è il C.mmp costruito dall'Università Carnegie-Mellon). Alcune possibili applicazioni sono l'uso di filtri differenti su un unico segnale oppure l'utilizzo di algoritmi diversi per decifrare lo stesso messaggio cifrato.

MIMD: **M**ultiple **I**nstruction, **M**ultiple **D**ata. Molti dei moderni parallel computer rientrano in questa categoria, ogni processore può eseguire un flow d'istruzioni diverso dalle altre ed inoltre, ogni processo può operare su un data flow differente. L'esecuzione può essere sincrona o asincorna, deterministica o non deterministica (Esempi: multiprocessori MIMD, architetture multi-core, molti dei supercomputer attuali)

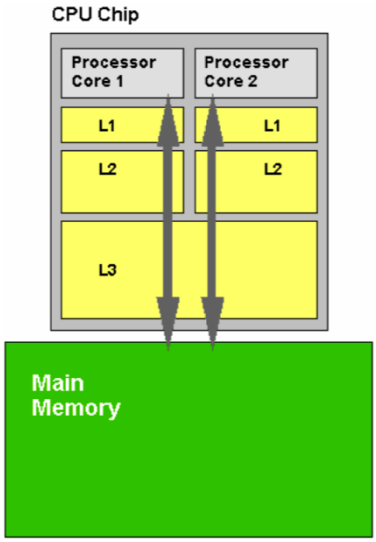
Diamo di seguito alcuni concetti di base:

- **Task**: un elemento di un'attività computazionale, una task è un programma o un insieme di istruzioni che sono eseguite da un processore
- **Parallel Tasks**: un iniseme di task che può essere eseguito da processori differenti nello stesso tempo, fornendo dei risultati corretti.
- **Sequential Execution**: l'esecuzione di un insieme di istruzioni una dopo l'altra, tutte le task sono caratterizzate da parti che devono essere eseguite sequenzialmente
- **Parallel Execution**: l'esecuzione di un programma per mezzo di task parallele. Ogni task può contenere lo stesso insieme di istruzioni oppure insiemi d'istruzioni differenti.

Le task parallele spesso hanno bisogno di scambiarsi dei dati, ci sono diverse strategie per farlo (ad esempio l'utilizzo di uno *shared bus* o l'utilizzo di un'*infrastruttura network*). Coordinare le attività parallele si ottiene per mezzo di **punti di sincronizzazione (synchronization points)** all'interno dell'applicazione. Una task non può procedere fino a quando un'altra task ha raggiunto un punto logico equivalente. La sincronizzazione spesso implica che almeno una task aspetta, questo porta ad una riduzione dell'efficienza nell'esecuzione di applicazioni parallele.



Le **CPU caches** sono piccoli pools di memoria che immagazzinano l'informazione di cui la CPU avrà bisogno successivamente, quale informazione è caricata nella cache dipende dagli algoritmi sofisticati ed alcune assunzioni sul codice da eseguire. L'obiettivo del cache system è di assicurare che la CPU abbia il prossimo bit di dati, di cui avrà bisogno, già caricato nella cache nel momento in cui va a cercarlo (chiamato anche *cache hit*). Più vicina è la cache alla CPU minore sarà il cache hit



La cache L1 è piccola e molto veloce, e molto vicina al core che la usa

La cache L2 è più grande e lenta, ed è usata da un singolo core.

La cache L3 è più comune con macchine multi-core, ed è più grande

e di conseguenza più lenta (condivisa dai core su una socket singola)

Nei computer paralleli con **memoria condivisa** tutti i processori vedono lo stesso spazio globale d'indirizzamento. I processori possono agire indipendentemente, ma condividono le stesse risorse di memoria. Se un processore modifica dei dati in memoria, tutti i processori lo vedono. Il modello corrispondente di programmazione impone che tutte le attività abbiano la stessa "vista" della memoria e siano autorizzate ad affrontare le stesse posizionei "logiche", indipendentemente da dove si trova fisicamente la memoria. PRO: Avere uno spazio d'indirizzi unico rende la programmazione più facile, la condivisione dei dati è più veloce ed uniforme, a causa della vicinanza dei processori. CONTRO: basso ridimensionamento, un incremento del numero delle CPU e delle memorie produce una crescita geometrica del traffico sui canali di comunicazione tra CPU e memorie, sul sistema che controlla la coerenza della cache e sulla gestione della cache stessa. Il programmatore deve inoltre gestire i meccanismi di sincronizzazione per garantire un accesso corretto ai dati.

Un **multiprocessore simmetrico (SMP)** è un computer system con processori mulitpli identici sulla stessa scheda madre che condividono la stessa memoria. Per via delle piccole dimensioni dei processori e della riduzione significativa dei requisiti per la banda del bus raggiunti dalle grandi cache, questo tipo di multiprocessori simmetrici sono estremamente convenienti, a condizione che esista una quantità sufficiente di larghezza di banda della memoria. Alcuni problemi di questa configurazione sono: la complessità nel configurare e mantenere questo tipo di sistema, non ridimensionabile e la migrazione di processi può portare a poco utilizzo della cache. Esempio: Quad CPU AMD opteron

Un **processore multi-core** ha unità d'esecuzione multiple sullo stesso chip, tutte condividenti la stessa memoria. I processori multi-core sono MIMD: core differenti eseguono thread differenti, operando su parti differenti della memoria. Esempi: Intel Xeon, IBM Cell (per play3)

Un **processore manycore** contiene numerosi (da una decina fino a migliaia) cores semplici e ed indipendenti. Sono molto efficienti per il computing parallelo ma hanno bassa performance sui single-thread. Esempi: CUDA e Intel Xeon Phi

La **memoria distribuita** richiede un network di comunicazione per lo scambio d'informazioni, ogni processore ha la sua memoria locale ed ogni memoria ha uno spazio d'indirizzi diverso. Le operazioni di scrittura e lettura sono locali, per cui non ci sono problemi di coerenza della cache. Per permettere ad una task di accedere ai dati remoti, il programmatore deve gestire esplicitamente la comunicazione tra le task. Il modello di programmazione corrispondente è chiamato **message passing (scambio di messaggi)**, definisce che ogni task può accedere direttamente solo alla sua memoria locale e deve comunicare con task remote per accedere a dati remoti. PRO: la memoria si ridimensiona con il numero di processori, ogni processore può accedere rapidamente alla sua memoria senza interferenze e sovraccarichi per la preservazione della cache coherence. Efficacia in termini di costi in quanto è possibile usare processori comuni. CONTRO: il programmatore è il responsabile di molti dettagli associati alla comunicazione tra i processori, può essere difficile dividere efficientemente i dati nelle memorie distribuite ed il tempo di accesso alle memorie può non essere uniforme.

Un **massively parallel processor (MPP)** è un computer singolo con molti processori networked, ognuno associato alla sua memoria privata. MPPs hanno molte caratteristiche comuni ai cluster ma hanno delle interconnessioni ad alta velocità specializzate al networking (mentre i cluster utilizzano gli hardware per il networking). Spesso gli MPPs tendono ad essere più grandi dei cluster.

Oggi, i computer più potenti utilizzano un approccio misto tra memoria distribuita e memoria condivisa: ogni nodo è un SMP con controllo della coerenza sulla cache, o un sistema multi core (o entrambi). I nodi sono connessi mediante un'infrastruttura framework. Il termine **network topology** si riferisce al modo in cui un insieme di nodi sono connessi tra di loro, le topologie network nascono nel contesto di architetture parallele ed algoritmi paralleli. Un **interconnection network** è un sistema di collegamenti che connettono uno o più dispositivi tra di loro per l'uso di comunicazioni inter-device. Uno **shared network** può avere al massimo un messaggio su di esso in qualsiasi momento (ad esempio un bus). Approccio opposto è lo **switched network** che permette messaggi point-to-point tra coppie di nodi e quindi supportra il trasferimento di più messaggi simultanei.

Per la rappresentazione dei network si utilizzano quadrati per rappresentare processori e/o memoria e cerchi per rappresentare gli switches. In una **direct topology** c'è essattamente uno swithc per ogni nodo processore, mentre in una **indirect topology** il numero di switches è più grande del numero di nodi processori. I **binary trees** sono sempre topologie indirette, che fungono da rete di commutazione per connettere una banca di processori l'uno all'altro. Il **2D mesh** è sempre usato come topologia diretta, con un processore attaccato ad ogni switch, anche gli **hypercube network** sono sempre topologie dirette. Per misurare le performance di un network si utilizzano

- **bandwith (banda/transmission rate)**: è la quantità di dati che può essere trasferita su una comunicazione per unità di tempo

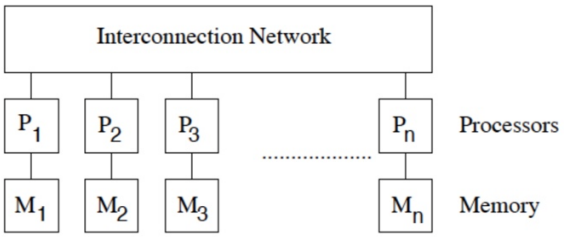
- **Latency (latenza/delay)**: è il tempo che un pacchetto di dati impiega a viaggiare da un punto ad un altro, la latenza diventa un problema solo quando sono necessari trasferimenti real-time .

ALGORITMI PARALLELI

Un approccio conveniente per progettare un *algoritmo sequenziale* consiste nel formularlo utilizzando un modello astratto di computazione chiamato **Random-access machine (RAM).**
In questo modello, la macchina è composta da un singolo processore connessa ad una memoria di sistema, ogni operazione basica della CPU (incluse le operazioni matematiche, logiche e gli accessi alla memoria) richiede un'unità di tempo. L'obiettivo del designer è quello di sviluppare un algoritmo con tempi e requisiti di memoria modesti. Il modello della macchina random access permette al designer di ignorare molti dettagli del computer su cui l'algoritmo verrà eseguito, ma comprende abbastanza dettagli che il designer può predire con un'accuratezza ragionevole su come l'algoritmo verrà eseguito.

La modellazione di **computazioni parallele** è più complicata rispetto a quella sequenziale in quanto i computer sono più complessi e variegati nell'organizzazione rispetto ai computer sequenziali. Come conseguenza, una gran porzione della ricerca sugli algoritmi paralleli ha approfondito il dibattito sul modeling esistono due classi di modelli:

- **Multiprocessor models**: una generalizzazione del modello sequenziale RAM, in cui è presente più di un processore. Esistono tre tipi base:
    - **Local memory machine model**, ogni processore può accedere alla sua propria memoria direttamente, ma per accedere alla memoria di un altro processore deve effettuare una richiesta attraverso il network. Come per il modello RAM, tutte le operazioni (inclusi gli accessi di memoria) richiedono un'unità di tempo. Il tempo per accedere alla memoria di un altro processore dipende dalla capacità del network di comunicazione e dal pattern degli accessi di memoria fatti dagli altri processori in quanto altri accessi potrebbero intasare il network.



    - **Modular memory machine model**, un processore accede alla memoria tramite un modulo di memoria inviando una richiesta attraverso il network. Tipicamente i processori ed i moduli di memoria sono organizzati in modo che il tempo per ogni processore di accedere ogni modulo sia strettamente uniforme. Così come in un *local memory machine model*, l'ammontare del tempo dipende dalla comunicazione del network e dal pattern dell'accesso alle memorie.
    - **Parallel random-access machine (PRAM) model**, un processore può accedere ogni parola della memoria in una singola unità di tempo, inoltre questi accessi avvengono parallelamente (quindi ogni processo può accedere alla memoria condivisa in un un'unità di tempo). Il **synchronous PRAM model** ha una somiglianza

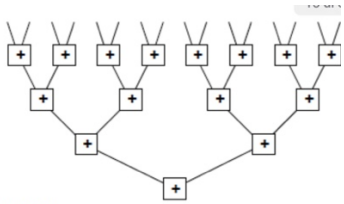
con l'esecuzione parallela su una macchina SIMD: tutti i processori eseguono lo stesso programma, lo stesso flusso di istruzioni PRAM in "lockstep", l'effetto delle operazioni dipende su dati locali e le istruzioni possono essere disattivate selettivamente. L'**asynchronous PRAM model** è composto da diversi programmi concorrenti e non ha il lockstep. Di seguito alcune proprietà del modello PRAM:

- **Exclusive Read**, in ogni step solo un processore può accedere ad una location nello stesso step.
- **Concurrent Read** nessuna restrizione sulla lettura.
- **Exclusive Write** in qualsiasi step, solo 1 processore può scrivere su una location durante lo stesso step
- **Concurrent Write** nessuna restrizione sulla scrittura -> non sicuro.

I pro del modello PRAM sono: l'avere a disposizione di un'astrazione lontana dal network e che evita di avere a che fare con i protocolli, ed inoltre molte idee vengono tradotte in altri modelli. Per quanto riguarda i contro: il modello di memoria è irrealistico (non si ha sempre lo stesso tempo costante per l'accesso), nella versione sincrona lockstep rimuove molta flessibilità e restringe le pratiche di programmazione, inoltre richiede un numero prestabilito di processori.

- **Work-depth models** sono modelli più astratti, non ci sono dettagli sulla dipendenza delle macchine per evitare di complicare il design e l'analisi degli algoritmi. Il costo di un algoritmo è determinato esaminando il numero totale di operazioni che esegue, e le dipendenze tra di esse. Un obiettivo importante è avere molti task con il minor numero di dipendenze possibili. Il **work W** di un algoritmo è il numero totali di operazioni che esegue, la sua **depth D** è la catena più lunga delle dipendenze tra le sue operazioni. Il **rapporto W/D** è il **parallelismo dell'algoritmo**.
  Il modello **DAG (Direct Acyclic Graph)** è un tipo speciale di work-depth model. È un grafo con archi orientati senza cicli, per modellare un algoritmo parallelo con DAG bisogna:
  1. Rappresentare piccoli segmenti d'istruzioni sequenziali come nodi del grafo, utilizzando pesi non negativi per mostrare i costi computazionali. 2. Usare gli archi solo per le dipendenze tra i nodi, come passaggio opzionale è possibile usare pesi non negativi sugli archi per mostrare i costi di comunicazione.

Direct acyclic graph (DAG) model



Example

ALGORITHM: SUM(A)
1 **if** $|A| = 1$ **then return** $A[0]$
2 **else return** SUM($\{A[2i] + A[2i+1] : i \in [0..|A|/2)\}$)

The *A[2i]+A[2i+1]* operations are performed in parallel, for each SUM() invocation.

W(n) = 2W(n/2)+1 where 1 is the constant work of the if
W(1) = O(1) constant time (may be larger than 1)
thus *W(n) = O(n)*

$D(n) = D(\lceil n/2 \rceil) + O(1) = O(\log n)$

Due operazioni A e B sono **ordinate** se esiste un cammino nel DAG da a A a B o da B ad A. Un **ordinamento sequenziale** di un algoritmo parallelo è l'esecuzione di istruzioni dell'algoritmo parallelo che non violano i limiti imposti dal DAG. Due operazioni sono **concorrenti** se non sono ordinate. La **concurrent read (CR)** consiste in due operazioni concorrenti dalla stessa memory location, **concurrent write (CW)** consiste in due operazioni concorrenti di scrittura sulla stessa memory location, **concurrent read-write (CRW):** due operazioni concorrenti accedono la stessa location, una effettua la lettura mentre l'altra la scrittura

La correttezza di un algoritmo parallelo è importante, se un algoritmo non funziona non esiste nessun punto da ottimizzare. Le *race condition* sono condizioni per cui l'ordine dell'esecuzione delle istruzioni incide sull'output dell'algoritmo. Se le istruzioni concorrenti provano a leggere e scrivere sulla stessa memory location, allora l'ordine in cui sono eseguite indica i valori che verranno scritti o letti dalle istruzioni concorrenti. Un algoritmo parallelo è *race free* se non è CW o CRW.

Per quanto riguarda la misura della **performance**, alcune relazioni da ricordare sono:

Lo **speed up**, $S_n = T_s/T_n$ con n pari al numero dei processor, T_s è il tempo d'esecuzione della versione sequenziale dell'algoritmo, T_n è il tempo d'esecuzione dell'algoritmo parallelo (idealmente S_n = n).

La **legge di Amdahl** afferma che $S_n = n/(f * n + (1 - f))$ con f che rappresenta la frazione dell'algoritmo che è strettamente sequenziale.

L'**efficiency** $E_n = S_s/n$ con n pari al numero dei processori ed S_n pari allo speed up, idealmente dovrebbe essere pari ad 1 mentre nel caso peggiore pari ad 1/n.

Un algoritmo parallelo è **work efficient** se W è asintoticamente lo stesso valore noto T_s per un algoritmo sequenziale per lo stesso problema.

Il **parallel Overhead** è il tempo necessario per coordinare le task parallele. Può includere i seguenti fattori: tempo d'inizializzazione dei task, tempo di sincronizzazione, comunicazione dei dati, spese di sovraccarico del software imposte da compilatori, librerie etc., tempo di completamento dell'attività.

La **granularità** è la misura qualitativa della relazione tra i processi e la comunicazione, la *granularità grossa* implica una quantità significativa di processi tra due eventi di comunicazione, mentre una *granularità fine* si ha quando avviene una quantità limitata di processi tra due eventi di comunicazione.

La **scalabilità** è l'abilità di un sistema parallelo di offrire un miglioramento delle performance proporzionale al numero di processori, i fattori che affliggono la scalabilità sono: la banda del bus che collega memoria e CPU, la banda ed il ritardo della rete, l'algoritmo parallelo e la sua implementazione, il *parallel overhead*. Considerando il formalismo work-depth, un algoritmo parallelo è **scalably parallel** se W/D → ∞ quando l'input size → ∞

Di seguito alcune linee guida per il **design di algoritmi paralleli**:

1. Posponi le considerazioni sui dettagli della piattaforma computazionale fino a dopo la fase di decomposizione, è improbabile che un progetto legato ad una particolare architettura funzioni bene su altre macchine, mentre un progetto astratto basato su attività e dipendenze può essere adattato per funzionare su qualsiasi macchina.
2. Progetta molti task indipendenti, il cui numero aumenta con la dimensione del problema

L'attività di **decomposizione** consiste nell'identificare i tasks e le loro dipendenze. Nella progettazione di algoritmi paralleli, ci sono molti pattern generali che possono essere usati su una vasta varietà di problemi. I pattern più importanti sono i seguenti:

- **Embarrasingly parallel**, questo modello si applica ai casi in cui la decomposizione in compiti indipendenti è *evidente*: ogni task fa il suo calcolo indipendente dalle altre attività ed il grafo dei task è completamente sconnesso. Nei metodi Monte Carlo, per esempio, un problema viene risolto conducendo numerosi esperimenti con variabili selezionate in modo pseudo-casuale. L'esempio illustrativo classico è il calcolo di pi-greco, un cerchio di raggio 1 è inscritto in un quadrato ed i punti sono scelti a caso all'interno del quadrato: l'area del cerchio, quindi pi-greco, può essere stimata come la frazione di punti che si trovano nel cerchio moltiplicata per l'area del quadrato. Questo problema può essere scomposto in compiti indipendenti che generano ciascuno un punto e determinano se si trova nel cerchio.
- **Divide-and-conquer**, in questo modello il problema è risolto ricorsivamente dividendo in due o più sotto-problemi dello stesso tipo fino a quando non diventano semplici abbastanza da essere risolti direttamente. Le soluzioni dei sotto-problemi sono poi unite per costruire una soluzione al problema originale. I sotto-problemi creati sono tipicamente indipendenti, quindi possono essere risolti in parallelo. La decomposizione dei tasks può essere decisa dinamicamente in run-tme, in modo da mantenere il miglior grado di parallelismo possibile. Affinché divide-and-conquer produca un algoritmo altamente parallelo, è spesso necessario parallelizzare il passaggio di dividere e il passaggio di unione dei sotto-problemi. (Esempio mergesort)
- **Data decomposition**: più che concentrarsi sulla decomposizione dei tasks, alcune volte è più facile ed efficiente scomporre le strutture dati. Successivamente, un'istanza differente dello stesso task è associata con ciascuna porzione. Ogni task svolge lo stesso lavoro, ma su dati diversi.
- **Randomization**: Numeri casuali sono usati in algoritmi paralleli per assicurare che i processori possono fare decisioni locali che, con alta probabilità, si sommano a buone decisioni globali. Esistono diversi usi della randomness, i principali sono:
    - *Sampling*: usa la randomness per selezionare una forma rappresentativa del campione da un insieme di elementi di input, per scopi di partizionamento o potatura
    - *Symmetry Breaking*: usa la randomness per selezionare un sott'insieme di operazioni indipendenti da un grande insieme di operazioni simmetriche, dove le

interdipendenze prevengono l'esecuzione simultanea di tutte le operazioni dell'insieme.

- *Load Balancing*: usa la casualità per assegnare i tasks ai processori.

MPI

La *memoria distribuita* richiede un network di comunicazione per lo scambio di informazioni; ogni processore ha la propria memoria locale ed uno spazio di indirizzamento separato. Le operazioni di lettura e scrittura sono locali, per cui non esistono problemi di coerenza della cache. Per permettere ai task di accedere ai dati remoti il programmatore deve esplicitamente controllare la comunicazione tra i task. Il modello di programmazione corrispondente è chiamato **message passing**, definisce che ogni task può accedere solo alla sua memoria locale e deve comunicare con altri task remoti per accedere ai dati remoti.

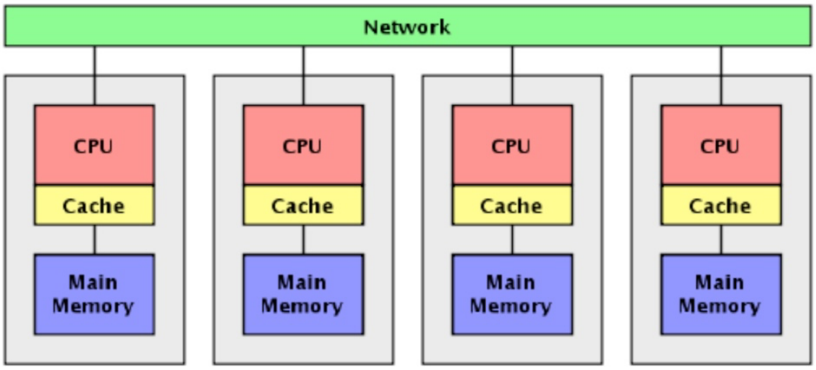
L'**MPI** è una specifica della libreria per il *message passing,* ovvero un modello per il message-passing ma non una specifica per il compilatore o un prodotto specifico. È *full featured*, progettato per permettere lo sviluppo di librerie di software paralleli e per garantire l'accesso ad hardware parallelo avanzato per gli utenti finali. Scrittori di librerie e sviluppatori di tools.

Bisogna sottolineare che MPI non è un'implementazione ma una specifica per le librerie, esistono numerose implementazioni dell'MPI. Non è un linguaggio e tutte le operazioni MPI sono espresse come funzioni, subroutines o metodi in base ai vari linguaggi (C, C++, Fortran.77 e Fortran-95 fanno parte degli standard MPI). Gli standard sono stati definiti attraverso un processo aperto dalla comunità dei vendors dei parallel computing, programmatori e sviluppatori di applicazioni.

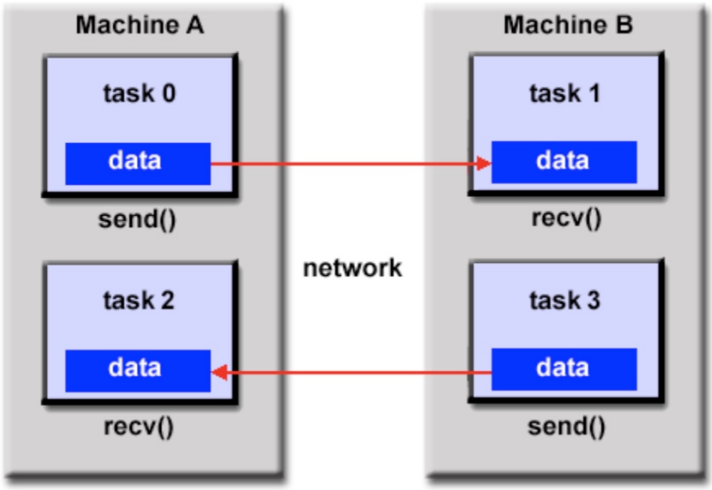
Alcuni obiettivi dell'MPI sono: progettare un'interfaccia di programmazione delle applicazioni (non necessariamente per i compilatori o una libreria di implementazione del sistema); permettere una comunicazione efficace, evitando la sovrapposizione di computazione con la comunicazione, evitare la copia memoria a memoria, e scaricare al coprocessore di comunicazione (se disponibile). Permettere implementazioni che possono essere usate su un sistema eterogeneo; assumere un'interfaccia di comunicazione affidabile: l'utente non ha bisogno di far fronte ad errori di comunicazione. Tali guasti sono trattati dal sottosistema di comunicazione sottostante. Definire un'interfaccia che può essere implementata su molte piattaforme dei fornitori, senza cambiamenti significativi nella comunicazione sottostante e nel software di sistema. La semantica dell'interfaccia dovrebbe essere indipendente dal linguaggio. L'interfaccia deve essere progettata per consentire la sicurezza dei thread.

La versione attuale è l'MPI 4.0, gli standard includono: comunicazione point-to-point, comunicazioni partizionate, datatypes, operazioni collettive, processi di gruppo, contesti di comunicazione, topologie di processi, gestione dei sistemi. L'oggetto Info, inizializzazione dei processi, creazione e gestione, comunicazione one-sided, I/O di file paralleli, strumenti di supporto, indicazioni di linguaggio per Fortran e C. NON È INCLUSO: operazioni che richiedono più supporto dal sistema operativo rispetto a quello attualmente standard, strumenti di costruzione

del programma, strutture di debug, **OpenMPI** è un'implementazione C/C++ di MPI, molto più pratica e portable di MPICH. Gli *hostfiles* sono semplici file di testo con gli hosts specificati, ogni host può specificare il numero di slot massimo di default da usare su quell'host (ovvero il numero di processori disponibili), i commenti sono supportati e le linee bianche ignorate.



La cooperazione tra processi è basata su *comunicazioni esplicite*: un processo **sender** ed un processo **receiver** scambiano i messaggi tra loro.



Ogni processo è un'istanza di un sub-programma in esecuzione, spesso lo stesso sub-programma è eseguito su data set differenti (ogni esecuzione diventa un processo). **SPMD:** single program, multiple data. Ogni processo è identificato da un numero intero, chiamato *rank*, che va da 0 ad n-1 dove n è il numero totale dei processi. E' possibile emulare MPMD utilizzando i costrutti if-then-else).

I **messaggi** sono composti da due parti:

- *envelope*
  - *source*: il rank del sender
  - *destination*: il rank del receiver
  - *tag*: l'ID del messaggio
  - *communicator*: il contesto della comunicazione
- *body*
  - *type*: MPI datatype

- *length*: il numero degli elementi
- *buffer*: array di elementi

Per quanto riguarda i **datatypes** distinguiamo: datatypes base (corrispondenti ai datatype standard di C o Fortran, come MPI_CHAR, MPI_INT, MPI_FLOAT, MPI_DOUBLE) e datatypes derivati (costruiti dai datatypes base o altri datatypes derivati, ad esempio MPI_SHORT, MPI_LONG, MPI_UNSIGNED_CHAR).

La **comunicazione point-to-point** è il tipo di comunicazione più semplice, comprende solo 2 processori (sender e receiver) e può essere sincrona o asincrona. Il **buffering** è implementato differentemente in ogni libreria MPI (lo standard non provvede ad una specifica).

Gli spazi di indirizzamento gestiti dai programmatori, per l'allocazione delle variabili, sono chiamati **application buffer**. Il **buffer di sistema** può esistere sia sul ricevitore che sul sender, è spesso limitato ed ha un comportamento imprevedibile, inoltre non può essere controllato dall'application programmer. MPI provvede anche ad un buffer di trasferimento gestito dall'utente.

Alcune operazioni possono causare il *blocco* di chi le chiama, le operazioni *non-bloccanti* permettono al processo di continuare subito dopo la chiamata. Il processo può *testare* o *aspettare* per la remote process completion (test), subito dopo la chiamata non-bloccante.

Le **operazioni bloccanti** possono avere: un send sincrono, un asincrono buffered send, un asincrono standard send, un ready send, un recv, sendrecv. Per le **operazioni non bloccanti** si hanno le stesse primitive, ma il sender non si blocca mai. È necessario controllare quando i buffers sono riutilizzabili, quando la comunicazione è terminata utilizzando le primitive test e wait.

La **comunicazione collective** impiega più di due processori, esistono diverse implementazioni. L'utilizzo di una **barriera** per la sincronizzazione dei processi: si aspetta che tutti raggiungono la barriera. Il **data movement** per le comunicazioni collettive: primitive broadcast, scatter, ghater. **Reduction** per le computazioni collettive, primitive: minimum, maximum, sum, logical OR, AND oppure definite dall'utente.

PROGRAMMARE IN MPI

MPI_ è un namespace riservato per le costanti MPI e le routines, dopo il prefisso solo la prima lettera è maiuscola. Tutte le funzioni MPI ritornano un codice intero, i nomi delle costanti sono maiuscoli. *mpi.h* è il nome del file header per C, mentre *mpif.h* è il nome standard del file header per Fortran. Gli header contengono le definizioni, le macros ed i prototipi delle funzioni che sono necessarie per compilare i programmi MPI. Un **MPI handle** è un language-agnostic name per comunicatori default, datatypes standard, etc.

Un **comunicatore** è un insieme di processi che possono comunicare tra di loro; ha un nome, una dimensione (numero dei processi), ogni processo può essere identificato univocamente, i processi

sono uguali. Due processi possono comunicare se appartengono allo stesso comunicatore, il comunicatore default è **MPI_COMM_WORLD** che include n processi, è un handle definito in mpi.h

Per conoscere il proprio rank un processo deve chiamare MPI_Comm_rank mentre per conoscere la dimensione del suo comunicatore deve chiamare MPI_Comm_size.

Per inizializzare il comunicatore default bisogna usare int MPI_Init(), per uscire dal sistema MPI int MPI_Finalize().

Ogni comando è eseguito da ogni processo indipendentemente, compilare produce un unico eseguibile e il sistema Runtime (open mpi) controlla come l'eseguibile è replicato su nodi di calcolo, come vengono creati i processi, come vengono gestiti l'output/l'errore standard.

COMUNICAZIONE POINT-TO-POINT

La funzione MPI_Send richiede:

- *buf*, è il puntatore al messaggio che deve essere mandato (application buffer).
- *count* è il numero di elementi nel messaggio.
- *Datatype* specifica il tipo di elementi nel messaggio.
- *Dest* è il rango del ricevitore.
- *Tag* è un intero non negativo il cui utilizzo è lasciato all'utente
- *comm*, ovvero il comunicatore

MPI_Recv richiede:

- *buf*, è il puntatore all'array su cui viene salvato il messaggio.
- *count* è il numero di elementi nel messaggio.
- *Datatype* specifica il tipo di elementi nel messaggio.
- *Source* è il rango del mittente.
- *Tag,* solo i messaggi con il tag specificato sono considerati per la ricezione.
- *comm*, ovvero il comunicatore
- *Status*, contiene le informazioni sull'envelope del messaggio da ricevere

Una comunicazione ha successo se e solo se: il sender specifica un rango del ricevitore valido, il ricevitore specifica un rank del source valido, sender e receiver sono nello stesso comunicatore, i tag corrispondono, i datatypes corrispondono e il receiver ha un buffer abbastanza largo per contenere il messaggio.

È possibile non specificare il sender (in MPI_Recv) utilizzando MPI_ANY_SOURCE, è anche possibile non specificare il tag usando MPI_ANY_TAG -> non è possibile farlo per il comunicatore. Entrambe vengono definite *wildcard*.

L'envelope di un messaggio è dato dalla struct MPI_RECV, che può essere ricavata dalla variabile *status.* È inoltre possibile accedere alla Source ed al tag, che in caso di wildcard, sono l'unico modo per conoscere il sender e qual è il tipo del messaggio ricevuto.

Per ottenere il numero di elementi con un datatype specifico nel messaggio ricevuto può essere chiamata la funzione MPI_Get_count sullo status.
I messaggi dalla stessa sorgente, sullo stesso comunicatore, con lo stesso tag allo stesso ricevitore arrivano nello stesso ordine in cui sono mandati. Il programmatore dovrebbe evitare la *starvation*: processo 0 e processo 1 mandano lo stesso messaggio al processo 2 che ha solo un MPI_Recv, di conseguenza solo un MPI_Send avrà successo.

I timer standard non sono adeguati per programmi MPI, in quanto non hanno un'accuratezza sufficiente e non sono portable. Il tempo d'esecuzione di un task è misurato ottenendo il tempo attuale prima che l'esecuzione inizi e dopo. MPI_Wtime() ritorna il tempo locale, misurato per un tempo prefissato. Per conoscere la risoluzione del timer bisogna usare MPI_Wtick().

Una comunicazione è **completata localmente** su un processo se il latter ha completato la sua parte di operazioni riferite alla comunicazione. Il completamento locale significa che il processo può eseguire le istruzioni

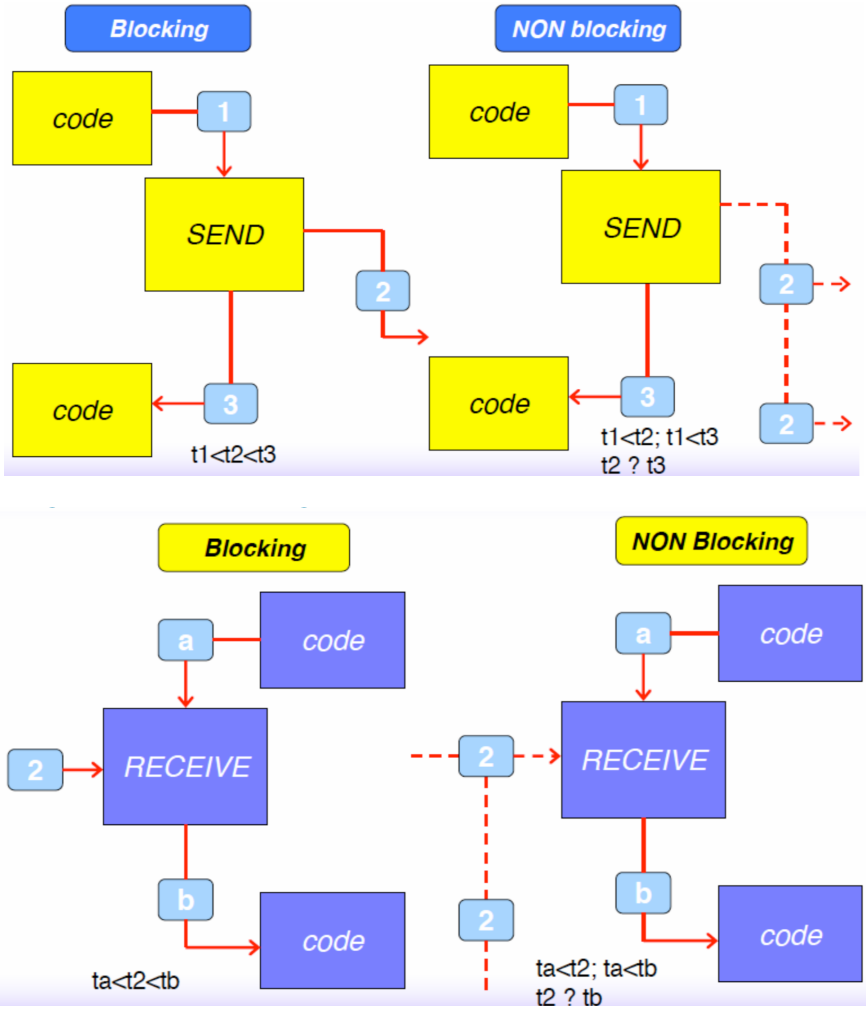
Una comunicazione è **completata globalmente** se tutti i processi coinvolti hanno completato le loro operazioni riguardanti la comunicazione: una comunicazione è completa globalmente se e solo se è completa localmente su tutti i processi coinvolti.

La fase di completamento della funzione SEND dipende dalle dimensioni del messaggio, buffered per dimensioni piccole mentre è sincrona per le dimensioni grandi. Nel primo caso il processo può uscire dal SEND dopo che il messaggio è stato copiato nel buffer system, nel secondo caso può uscire dal SEND solo quando una funzione RECV viene eseguita. Riepilogando abbiamo i seguenti criteri di completamento:

- *Send sincrona*: completata quando l'application buffer può essere riusato e la ricezione del messaggio è cominciata.
- *Send buffered*: completata quando il messaggio è stato copiato completamente sul transfer buffer.
- *Send standard*: completata quando l'application buffer può essere riutilizzato.
- *Receive*: completata quando il messaggio è arrivato.

Le modalità di comunicazioni **blocking** ritornano il control al processo che ha invocato la primitiva di comunicazione solo quando il latter ha completato.

Le modalità di comunicazione **nonblocking** ritornano immediatamente control al processo che invocato la primitiva di comunicazione, un controllo sull'effettivo completamento della comunicazione deve essere eseguito in seguito; nel frattempo, il processo può eseguire altre operazioni.



Per quanto riguarda le **funzioni di comunicazione bloccanti** abbiamo:

- *Synchronous send*, MPI_Ssend: manda il messaggio e resta bloccato fino a quando l'application buffer del sender è pronto per essere usato per un'altra operazione *e* il ricevitore ha iniziato a ricevere il messaggio. Pro: questa send è più sicura di quella standard, perché il network non è sovraccaricato con messaggi sospesi, inoltre sender e ricevitore sono sempre sincronizzati (il programma parallelo risultate è più deterministico) Contro: rischio di tempo di idle e deadlocks.
- *Buffered send, MPI_Bsend*: è completato immediatamente appena il processo ha copiato il messaggio sul transfer buffer che deve essere controllato esplicitamente dal programma mediante le funzioni MPI_Buffer_attach per allocare l'aria di memoria sul transfer buffer e MPI_Buffer_detach per de-allocare l'aria di memoria sul transfer buffer. In questo caso sender e receiver non sono sincronizzati.
- *Standard send, MPI_Send*: completata quando il messaggio è stato mandato e l'application buffer può essere riutilizzato. Il messaggio può restare nella rete di comunicazione per un po'.

- *Ready send, MPI_Rsend*: una send che usa la modalità ready communication può essere inizializzata solo se la ricezione corrispondente è già stata pubblicata, altrimenti l'operazione è errata e il suo risultato è indefinito. (Da evitare)
- *Send-Receive, MPI_Sendrecv*: esegui un send bloccante e un'operazione di receive. Entrambe usano lo stesso communicator, ma con possibili tag differenti. Il send buffer e receive buffer devono essere disjoint, e possono avere lunghezze e datatypes differenti.

Una comunicazione non bloccante è tipicamente composta da tre fasi:

1. Iniziare un send o un receive del messaggio
2. Eseguire un'attività che non implica l'accesso del dato interessato nella comunicazione
3. Attendere il completamento della comunicazione

Pro: miglioramento delle performance in quanto permette la sovrapposizione di fasi di comunicazione con fasi di computazione, riducendo gli effetti della latenza delle comunicazioni. Inoltre, evitano situazioni di **deadlock**. Contro: programmare il passaggio di messaggi non bloccanti è più complicato.

- *Synchronous nonblocking send*: *MPI_Issend* (la I sta per immediate) ed *MPI_Wait.* il buf non può essere usato tra Issend ed il wait, l'issend seguito immediatamente da un wait è un Ssend. Lo status non è usato nell'Issend ma nel wait, wait e test permettono di saper quando il ricevitore ha ricevuto il messaggio.
- *Standard nonblocking send*: *MPI_Isend e MPI_Comm,* inizializza l'operazione di send ed identifica un'area di memoria che deve essere usata come buffer. L'esecuzione prosegue senza aspettare che il messaggio sia copiato sull'application buffer. Un programma non può modificare l'application buffer fino a quanto una chiamata successiva a MPI_Wait o MPI_Test non indica che la send è stata completata. Aumenta il grado di sovrapposizione tra computazione e comunicazione.
- *Nonblocking Receive: MPI_Irecv, MPI_Comm*. Inizializza un'operazione di ricevimento, identifica un'area di memoria come buffer di ricevimento. La funzione ritorna immediatamente, senza aspettare che il messaggio sia copiato interamente sull'application buffer locale ma non può essere usato quando la request è pending (per sapere quando ha finito bisogna usare la funzione wait o test). Per controllare lo status di una richiesta di ricevimento bisogna usare il request handle che è richiesto dalla funzione.

Quando si usano comunicazioni *Point-to-Point* è fondamentale controllare che una comunicazione sia stata completata prima di usare receive buffer o send buffer. MPI offre due tipi di controlli: WAIT che permette di fermare l'esecuzione del processo fino a quando la comunicazione non è stata completata oppure TEST che ritorna al processo che chiama un valore che è TRUE se le comunicazioni sono state completate, FALSO altrimenti.

Supponiamo di avere un MPI_Ssend ed MPI_Recv in serie che chiamano due processi diversi, in questo caso si ha una situazione di **Deadlock** in quanto Ssend non completa fino a quando il Recv corrispondente non è iniziato, ma tutti i processi sono bloccati su Ssend e quindi nessuno inizia il

Recv. Per evitare i deadlocks occorre *cambiare l'ordine delle chiamate (*pericoloso in caso di dipendenze tra più di due processi, occorre una buona conoscenza del pattern di comunicazioni che deve essere deterministico), utilizzare *operazioni non bloccanti* (aumenta la complessità del programma) ed usare *buffered modes* (aggiungendo determinismo al programma senza aumentarne di troppo la complessità).

È possibile che molte comunicazioni non bloccanti sono "postate" nello stesso tempo, per questi casi MPI provvede a delle operazioni per il controllo simultaneo del loro completamento: MPI_*Waitany/TestAny* (per aspettare o testare il completamento di un messaggio nella lista), *MPI_Waitall/MPI_Testall* (per aspettare o testare il completamento di tutti i messaggi), *MPI_Waitsome/MPI_Testsome* (per aspettare alcuni messaggi).

DATATYPES DERIVATI

I **datatypes derivati** sono usati per messaggi polimorfici, ovvero messaggi che sono caratterizzati da elementi composti di vari datatypes generici (tutti in un singolo blocco). I datatypes derivati sono usati anche per trasferire dati da buffers non contigui (come una porzione di una matrice). Per questo motivo MPI permette ai programmatori di specificare buffers di comunicazioni contigui o mixed, in modo che oggetti di varie forme e dimensioni possono essere trasferiti direttamente senza essere copiati.

Per specificare dati non contigui di un singolo datatype oppure dati contigui di un tipo misto è possibile utilizzare alcune soluzioni: creare chiamate MPI multiple per mandare e ricevere ogni elemento dei dati, utilizzare MPI_Pack per combinare datatype differenti in una memoria contigua per mandare ed utilizzare MPI_Unpack per estrarre i dati in una memoria non contigua dopo essere stati ricevuti oppure usare MPI_BYTE per evitare le regole di datatype-matching. Queste soluzioni sono lente e consumano memoria. Utilizzare MPI_Pack o MPI_Byte può risultare in un programma che non è portabile su un sistema eterogeneo di macchine.

I datatypes derivati sono creati in run-time prima di essere usati in una comunicazione per mezzo del costruttore appropriato, che popola un descrittore di tipo datatype (MPI_Datatype). Abbiamo:

- *MPI_Type_contigous*: è il datatype derivato più semplice, un insieme di elementi contabili dello stesso tipo
- *MPI_Type_vector*: richiede la lunghezza del nuovo blocco (quanti elementi vanno in un blocco), uno stride (ovvero il numero di step tra due blocchi di elementi) ed infine il count che indica il numero di blocchi.
- *MPI_Type_struct*: richiede il numero di blocchi (count), il numero di elementi di ogni blocco (array), il numero di displacement-byte di ogni blocco (array, per calcolarlo occorre fare la differenza tra l'address di memoria del blocco i con l'address di memoria del blocco 0 ottenuti tramite la funzione MPI_Get_address), il tipo di elementi in ogni blocco (array di datatypes semplici oppure oggetti), ed infine il nuovo datatype (handle).

Un datatype creato deve essere confermato mediante la funzione MPI_Type_commit, dopo essere usato il datatype deve essere dislocato mediante la funzione MPI_Type_free.

Ogni datatype ha una dimensione (ovvero il numero di byte che devono essere trasferiti) ed un extent (l'intervallo tra il primo e l'ultimo byte) ottenibili mediante MPI_Type_size ed MPI_Type_get_extent.

COMUNICAZIONI COLLETTIVE

MPI provvede funzioni che implementano pattern di comunicazioni che coinvolgono più processi, per evitare al programmatore di implementarli mediante comunicazioni Point-to-Point. Esistono tre classi: all-to-one, one-to-all ed all-to-all. Le operazioni sono collettive: tutti i processi devono comunicare, quindi, chiamare la routine collettiva con gli stessi argomenti, non esistono tag e i buffers di ricevimento devono combaciare con la dimensione esatta del messaggio.

La funzione MPI_Barrier implementa una *barriera di sincronizzazione*, spesso non è necessaria perché i processi sono sincronizzati attraverso la comunicazione (ad esempio un processo non può avanzare senza avere tutti i dati).
MPI_Bcast implementa un *broadcast* (one-to-all class): replica il contenuto del buffer radice a tutti i processi mediante il comunicatore, non implica sincronizzazione necessariamente. I parametri sono: l'indirizzo del buffer di ricevimento/mandante, il numero di elementi che formano il dato da mandare, il tipo di dato che viene mandato, il rango del sender e il nome del comunicatore.

MPI_Scatter

Usando MPI_Gather (all-to-one class) tutti i processi (incluso quello radice) mandano il contenuto del loro send buffer al processo radice, il processo radice riceve i dati e gli ordina in base al rango del mandante.

MPI_Reduce per effettuare una riduzione, usate per compiere operazioni che coinvolgono dati distribuiti su un gruppo di processi. (somma e prodotto globale, massimo e minimo globali, operazioni globali definite dall'utente etc.). Se ci sono più di due task, MPI costruisce un albero binario delle task comunicanti.

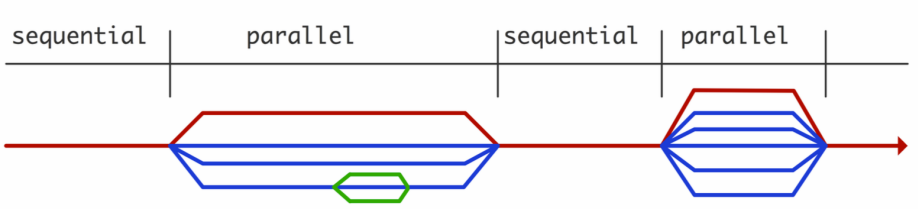
Nel caso di **comunicazioni collettive non bloccanti**, tutte le chiamate sono locali e ritornano immediatamente indipendentemente dallo stato degli altri processi. La chiamata inizializza l'operazione, che indica che il sistema può iniziare la copia dei dati dal send buffer al receiver buffer. Una volta iniziato, tutti i send buffers ed i buffer associati agli argomenti di input non dovrebbero essere modificati, e non si dovrebbe accedere a tutti i receive buffer fino alla fine delle operazioni collettive. La chiamata ritorna una **request handle** che deve essere passata per completare la chiamata. (funzioni Ibarrier, Ibcast, Iscatter, Igather)

OPEN-MPI

Nei computer paralleli con **memoria condivisa**, tutti i processori vedono lo stesso spazio d'indirizzo globale. I processi possono agire indipendentemente, ma condividono le stesse risorse di memoria. Se un processore modifica i dati in memoria, tutti gli altri processori vedono le modifiche. Il modello di programmazione corrispondente impone che tutti i task abbiano la stessa "vista" della memoria e sono autorizzate ad affrontare le stesse posizioni "logiche", indipendentemente da dove si trova fisicamente la memoria.

OpenMP aggiunge costrutti per **threading shared-memory** su C, C++, Fortran. OpenMP consiste in direttive per il compilatore che può includere clausole, Runtime-routine e variabili di sviluppo.

Inizia l'esecuzione come un singolo processo (thread master) e comincia un costrutto parallelo (usando direttive speciali), il thread master crea quindi una squadra di thread. Il modello dell'esecuzione parallela fork-join è quello in figura:



DI default il numero di thread coincide con il numero di processori del nodo, per controllare il numero di thread usato da un programma OpenMP è possibile impostare la variabile di sviluppo "OMP_NUM_THREADS", il numero di thread può essere imposto con una routine "omp_set_num_threads"

Le variabili che si trovano fuori da una regione parallela sono *condivise*, mentre le variabili che si trovano all'interno di una regione parallela sono *private* (allocate nello stack del thread). I programmatori possono modificare questo aspetto attraverso le clausole "private()" e "shared()", è una responsabilità del programmatore di assicurare che thread multipli accedono propriamente alle variabili condivise. Una **direttiva** è un statement, OpenMP può includere delle clausole. Le direttive principali sono Fork e Barrier. Una *regione parallela* è un blocco di codice che sarò eseguito da thread multipli. E' il costrutto fondamentale di OpenMP. La direttiva *for* richiede che lo statement successivo è un loop for, rende il l'indice del loop privato per ogni thread ed esegue un sott'insieme di iterazioni in ogni thread. (#pragma omp for), lascia la decisione di allocazione dati al compilatore (è possibile specificarla manualmente tramite "#pragma omp for schedule(…)".

La direttiva *single* specifica che il codice incluso deve essere eseguito solo da un thread nel team, la direttiva *master* indica che la regione che deve essere eseguita solo dal master thread del team.

La direttiva *sections* per la condivisione del lavoro è seguita da una regione che ha una direttiva *section* per ogni task, c'è una barriera implicita alla fine di una direttiva sections.

Per eseguire due operazioni X ed Y simultaneamente in parallelo non deve esserci nessun dato condiviso che è letto da X e scritto da Y, e non deve esserci nessun dato condiviso che è scritto da X e scritto da Y.

La direttiva *critical* specifica una regione di codice che deve essere eseguita solo da un thread alla volta, la clausola *reduction* rende la variabile specificata privata per ogni thread e combina i risultati privati su di essa

CUDA

Le schede grafiche (**GPU – G**raphic **P**rocessor **U**nit) si sono evolute negli ultimi anni in processori manycore multithread altamente parallelizzati con un enorme potenza di calcolo e larghezza di banda di memoria molto elevata. La ragione dietro la discrepanza tra abilità floating point tra la CPU e la GPU è che la GPU è specializzata in computazione intensiva altamente parallela. Le GPU sono particolarmente adatte per affrontare problemi:

- Che possono essere espressi come calcoli paralleli di dati oppure se lo stesso programma è eseguito su molti elementi di dati in parallelo.
- Con alta intensità aritmetica, il rapporto tra operazioni aritmetiche e operazioni di memoria.
- Con requisiti inferiori per un sofisticato controllo del flusso
- Per cui non c'è bisogno di cache di grandi dimensioni

L'elaborazione parallela dei dati mappa gli elementi dei dati in thread di elaborazione paralleli. Molti algoritmi al di fuori del campo del rendering e dell'elaborazione delle immagini sono accelerati dall'elaborazione parallela dei dati, dall'elaborazione generale del segnale o dalla simulazione fisica alla finanza computazionale o alla biologia computazionale.

CUDA è introdotto da NVIDIA nel 2006, insieme ad un ambiente software (**CUDA toolkit**) che permette agli sviluppatori di usare C/C++ come linguaggio di programmazione ad alto livello. Il CUDA toolkit fornisce un ambiente di sviluppo completo per gli sviluppatori C e C++ per creare applicazioni accelerate da GPU. Il CUDA Toolkit include un compilatore per le GPU NVIDIA (**nvcc**), librerie matematiche ed uno strumento per il debugging e l'ottimizzazione delle performance delle applicazioni (**cuda-gdb**).

Nvcc è un compilatore che offre delle opzioni di command lines semplici e familiari, gli esegue invocando la collezione degli strumenti che implementano stage diversi della compilazione. I file sorgente possono includere un mix di *codice host* (il codice eseguito dall'host) e *codice device* (il codice che viene eseguito sul dispositivo). Il workflow base di nvcc consiste in dividere il codice device da quello dell'host e compilare il codice device in forma assembly o binaria. Il front end dei processi del compilatore dei source file di CUDA seguono le regole di sintassi C/C++, C++ è supportato per l'host code e per il device code (con alcune restrizioni). *Thrust* è un template della libreria C++ per CUDA basata su STL (standard template library). Thrust permette di implementare

applicazioni altamente performanti con minimo sforzo in termini di programmazione attraverso un'interfaccia ad alto livello altamente interoperabile con CUDA C.

La sfida consiste nello sviluppare applicazioni software che possono scalare il loro parallelismo con l'aumentare del numero di cores dei processori, molte delle applicazioni grafiche 3D scalano trasparentemente il loro parallelismo per GPUs manycore con un'ampia varietà di numeri di cores. Il modello di programmazione parallela di CUDA è progettato per superare questo problema rimanendo semplice per i programmatori che sono familiari con linguaggi di programmazione standard come C.

Al suo centro ci sono tre astrazioni chiave:

- Una gerarchia di gruppi thread
- Memorie condivise
- Barrier Synchronization

Che sono esposti semplicemente al programmatore come insieme minimo di estensioni del linguaggio. Queste astrazioni guidano il programmatore nel partizionare il problema in sotto-problemi che possono essere risolti indipendentemente in parallelo, e poi in pezzi più fini risolvibili cooperando in parallelo. Un programma CUDA compilato può quindi essere eseguito su un qualsiasi numero di core del processore, e solo il sistema di Runtime deve conoscere il numero fisico del processore. Attualmente sono supportate due interfacce per scrivere programmi CUDA: **CUDA C** (insieme d'estensioni minime per il linguaggio C) e **CUDA driver API** (un'API low-level C), sono mutualmente esclusivi -> un programma deve usare solo uno dei due.

CUDA C è stato rilasciato con un API Runtime, che è costruita sul CUDA driver API. Sia l'API Runtime che il driver API provvedono funzioni per allocare e dislocare la memoria del dispositivo, trasferire dati tra memoria dell'host e la memoria del dispositivo, controllare sistemi con dispositivi multipli etc. Inizializzazione, contesto, e gestione dei moduli sono tutti impliciti ed il codice risultante è più coinciso, al contrario, CUDA driver API richiede più codice, è più difficile da programmare e debuggare ma offre un miglior livello di controllo ed è indipendente dal linguaggio in quanto gestisce codice binario o assembly.

CUDA C estende C permettendo al programmatore di definire funzioni in C, chiamate **kernels,** che sono eseguite N volte in parallelo da N threads di CUDA differenti. Un kernel è definito usando la dichiarazione ___global___ e il numero di threads CUDA per ogni chiamata è specificato usando la sintassi <<<…>>>. A ogni thread che esegue il kernel viene assegnato un *indice di thread* accessibile all'interno del kernel tramite la variabile *threadIdx* incorporata. Per convenienza, *threadIdx* è un vettore di 3 componenti, in modo che i threads possono essere identificati usando un indice monodimensionale, bidimensionale o tridimensionale di indici thread formando dei thread block (che possono essere monodimensionali, bidimensionali, tridimensionali).

Questo porta ad un modo naturale di invocare calcoli tra gli elementi in un dominio come vettori, matrici o campi. *ThreadID* è un numero scalare unico che identifica per ogni thread unicamente il

threadblock a prescindere dalla sua dimensione (1D,2D,3D). Il thread index ed il threadID si riferiscono l'uno all'altro in modo semplice:

1. Per il blocco 1D sono uguali
2. Per il blocco 2D di dimensione (Dx,Dy), il threadID del thread di indice (x, y) è (x + y Dx)
3. Per il blocco 3D di dimensione (Dx,Dy,Dz), il threadID è il thread di indice (x, y, z) è (x + y Dx + z Dx Dy)

In questo modo è possibile ordinare i threads incrementando gli ID.

I thread all'interno di un blocco possono cooperare condividendo i dati attraverso una memoria condivisa per blocco e sincronizzando la loro esecuzione per coordinare gli accessi alla memoria. Si possono specificare i punti di sincronizzazione nel kernel chiamando la funzione intrinseca *__syncthreads()* che agisce da barriera in cui tutti i thread nel blocco devono attendere prima che sia consentito procedere. Per una cooperazione efficiente, la memoria condivisa dovrebbe essere una memoria a bassa latenza vicino a ciascun core del processore (come una cache L1). *__syncthreads()* dovrebbe essere leggero e tutti i thread di un blocco dovrebbero risiedere sullo stesso core del processore. Il numero dei threads su un blocco è quindi limitato dalle limitate risorse di memoria di un core del processore. Sulle GPU attuali un thread block può contenere fino a 1024 threads.

Un kernel può essere eseguito da thread blocks multipli con la stessa forma, in questo modo il numero totale di threads è dato dal numero di thread per blocco moltiplicato per il numero di blocchi. Questi blocchi multipli sono organizzati in griglie di thread blocks (1D o 2D). La dimensione della griglia è specificata dal primo parametro della sintassi <<<…>>>, ogni blocco della griglia può essere identificato da un indice 1D o 2D accessibile all'interno del kernel attraverso la variabile *blockIdx* incorporata.

La dimensione del blocco del thread è accessibile dal kernel tramite la variabile *blockDim* incorporata. Un thread block di dimensione 16 x 16 (256 threads) è una scelta comune, la griglia è creata con abbastanza blocchi da avere un thread per ogni elemento della matrice. I thread block devono essere eseguiti in modo indipendente: deve essere possibile eseguirli in qualsiasi ordine, in parallelo o in serie. Questo requisito di indipendenza consente di programmare i blocchi di thread in qualsiasi ordine su qualsiasi numero di core, consentendo ai programmatori di scrivere codice che scala con il numero di core. Il numero di blocchi di thread in una griglia è tipicamente dettato dalla dimensione dei dati in fase di elaborazione piuttosto che dal numero di processori del sistema, che può essere di gran lunga più grande.

Il modello di programmazione CUDA prevede che i threads CUDA sono eseguiti su un dispositivo separato fisicamente, che opera come coprocessore per l'host che esegue il programma C. Questo è il caso, per esempio quando i kernel vengono eseguiti su una GPU e il resto del programma viene eseguito su una CPU. Sia l'host che il supporto devono avere la propria DRAM (*host memory* e *device memory*).

I threads CUDA possono accedere ai dati da diversi spazi di memoria durante la lora esecuzione, ogni thread ha una *memoria privata locale* (massimo 255 registri), ogni blocco di thread ha una *memoria condivisa* (64 kb) visibile a tutti i thread del blocco e con lo stesso tempo di vita del blocco, infine tutti i thread hanno accesso alla stessa *memoria globale* (12-40 GB). Ci sono due spazi di memoria addizionali read-only accessibili da tutti i thread: il *constant* ed il *texture* memory spaces. Gli spazi di memoria globali, costanti e texture sono ottimizzati per diversi usi di memoria, la memoria texture offre anche diverse modalità di indirizzamento per alcuni formati di dati specifici ed inoltre sono persistenti tra i lanci del kernel dalla stessa applicazione.

La memoria globale è tipicamente allocata usando *cudaMalloc()* e liberata usando *cudaFree()*, i trasferimenti di dati tra memoria host e memoria globale avvengono mediante la funzione *cudaMemcpy()*. Le seguenti operazioni del dispositivo sono asincrone rispetto all'host

- Avvii di Kernel
- Copie di memoria su una singola memoria del dispositivo
- Copie di memoria dall'host ad un device con memory block di 64 KB o inferiori.
- Copie di memoria eseguite da funzioni che hanno come suffisso *Async*
- Chiamate alle funzioni sul memory set

Tutte le funzioni Runtime ritornano un codice d'errore, ma per le funzioni asincrone questo codice non può riportare nessuno degli errori asincroni che possono avvenire sul dispositivo in quanto la funzione ritorna prima che il device ha completato il task. L'unico modo per controllare gli errori asincroni subito dopo la chiamata ad una funzione asincrona consiste nel sincronizzare subito dopo la chiamata chiamando la funzione *cudaDeviceSynchronize()* e controllando il codice restituito da quella funzione.

In Runtime viene sempre mantenuta una variabile di errore per ogni thread host che viene inizializzata in *cudaSuccess* e viene sovrascritta dal codice di errore ogni volta che si verifica un errore.

Il **parallelismo dinamico** è un'estensione del modello di programmazione CUDA che permette ad un kernel CUBA di creare e sincronizzare il nuovo lavoro direttamente sulla GPU (kernel all'interno dei kernel). L'abilità di creare lavoro direttamente dalla GPU può ridurre il bisogno di trasferire il controllo dell'esecuzione e dei dati tra host e dispositivo, come configurazione di lancio le decisioni possono essere prese in Runtime dai threads che vengono eseguiti sui dispositivi In aggiunta, il lavoro parallelo data-dependent può essere generato in linea all'interno di un kernel in run-time, sfruttando dinamicamente gli scheduler hardware ed i bilanciatori di carico della GPU e in risposta a decisioni o carichi di lavoro basati sui dati.

Un thread device che configura ed avvia una nuova griglia appartiene alla griglia padre e la griglia creata dall'invocazione è una griglia figlio. L'invocazione ed il completamento delle griglie figlie è correttamente nidificato, il che significa che la griglia padre non è considerata completa fino a quando tutte le griglie figlie create dal suo thread non sono state completate, ed il Runtime garantisce una sincronizzazione implicita tra il genitore ed il figlio. Le operazioni di CUDA Runtime

da qualsiasi thread, compresi i lanci del kernel, sono visibili su tutti i thread appartenenti ad una griglia.

CUDA streams ed **events** permettono il controllo sulle dipendenze tra grid e lanci: le griglie lanciate nello stesso flusso vengono eseguite in ordine e gli eventi possono essere utilizzati per creare dipendenze tra i flussi. I flussi e gli eventi creati sul device servono esattamente allo stesso scopo, con più thread nella stessa griglia che si avviano nello stesso flusso, l'ordine interno del flusso dipende dalla pianificazione dei thread all'interno della griglia (può essere controllata mediante primitive di sincronizzazione come __syncthreads(). Il Runtime del dispositivo è un sott'insieme funzionale del Runtime host. La gestione del dispositivo a livello di API, l'avvio del kernel, il memcpy del dispositivo, la gestione dei flussi e la gestione degli eventi sono esposti dal Runtime del device. La sintassi e la semantica del Runtime del device sono in gran parte uguali quelle dell'host. Un programma di Runtime del device può essere compilato e collegato in un unico passaggio, se tutti i file di origine richiesti possono essere specificati dalla riga di comando, è anche possibile compilare prima i file di origine .cu CUDA in file oggetto, e poi collegarli insieme in un processo in due fasi.

Le griglie padre e figlio condividono la stessa memoria globale e costante, ma hanno memoria locale e condivisa distinta. C'è solo un punto di tempo nell'esecuzione di una griglia figlio in cui la sua vista della memoria è pienamente coerente con il thread padre: nel punto in cui la griglia figlio viene invocata dal padre. Tutte le operazioni di memoria globale nel thread padre prima dell'invocazione della griglia figlio sono visibili nella griglia figlio. L'unico modo, per la griglia padre, per accedere alle modifiche apportate dai thread nella griglia figlio è tramite un kernel lanciato nel flusso *cudaStreamTailLaunch.*

La memoria condivisa e la memoria locale è privata rispettivamente per un blocco di thread o thread, non è visibile o coerente tra padre e figlio. Il comportamento non è definito quando un oggetto in una di queste posizioni viene referenziato al di fuori dell'ambito a cui appartiene e può causare un errore. Il compilatore NVIDIA tenterà di avvertire se può rilevare che un puntatore alla memoria locale o condivisa viene passato come argomento all'avvio di un kernel. In fase di esecuzione, il programmatore può utilizzare l'estrinsecco __isGlobal() per determinare se un puntatore fa riferimento alla memoria globale e quindi può essere passato in modo sicuro a un lancio figlio. La **memoria unificata** è un componente del modello di programmazione CUDA che definisce uno spazio di memoria gestito in cui tutti i processori (CPU e GPU) vedono una singola immagine di memoria coerente con uno spazio di indirizzi comune. Il sistema sottostante gestisce l'accesso ai dati e la localizzazione all'interno di un programma CUDA senza la necessità di chiamate di copia della memoria esplicita. Questo avvantaggia la programmazione GPU in due modi principali:

- La programmazione GPU è semplificata unificando gli spazi di memoria in modo coerente tra tutte le GPU e le CPU del sistema e fornendo un'integrazione del linguaggio più stretta e diretta per i programmatori CUDA.

- La velocità di accesso ai dati è massimizzata migrando in modo trasparente i dati verso il processore che lo utilizza.

L'unificazione degli spazi di memoria significa che non c'è più bisogno di trasferimenti espliciti di memoria tra host e dispositivo. Qualsiasi allocazione creata nello spazio di memoria gestito viene automaticamente migrata dove è necessaria. La memoria unificata tenta di ottimizzare le prestazioni della memoria migrando i dati verso il dispositivo a cui si accede (cioè, spostando i dati nella memoria host se la CPU vi sta accedendo e nella memoria del dispositivo se la GPU vi accederà). La migrazione dei dati è trasparente per un programma, il sistema cercherà di posizionare i dati nella posizione in cui è possibile accedervi in modo più efficiente senza violare la coerenza. I dispositivi con capacità di elaborazione inferiore a 6.x non possono allocare più memoria gestita rispetto alle dimensioni fisiche della memoria GPU. I dispositivi di capacità di elaborazione 6.x estendono la modalità di indirizzamento per supportare l'indirizzamento virtuale a 49 bit che è abbastanza grande da coprire gli spazi di indirizzi virtuali a 48 bit delle CPU moderne, così come la memoria della GPU. L'ampio spazio degli indirizzi virtuali e la capacità di errore di pagina consentono alle applicazioni di accedere all'intera memoria virtuale del sistema, non limitata dalle dimensioni della memoria fisica di un qualsiasi processore. Ciò significa che le applicazioni possono sottoscrivere eccessivamente il sistema di memoria: in altre parole possono allocare, accedere e condividere array più grandi della capacità fisica totale del sistema, consentendo l'elaborazione out-of-core di set di dati molto grandi. Ogni dispositivo CUDA è un array scalabile di **Streaming Multiprocessors (SM**), ognuno dei quali include N core (in effetti, sono solo ALU). Ogni multiprocessore crea, gestisce, pianifica ed esegue thread in gruppi di N thread paralleli (1 per ogni core) chiamati *warps*, l'SM impiega un'**architettura** chiamata **SIMT** (istruzione singola, thread multiplo). Il multiprocessore mappa ogni thread a un nucleo e ogni thread viene eseguito in modo indipendente con il proprio indirizzo di istruzione e lo stato di registro. La capacità di calcolo di un dispositivo è definita da un numero di revisione maggiore X e da un numero di revisione minore Y ed è denotata dalla coppia X.Y. I dispositivi con lo stesso numero di revisione principale sono della stessa architettura di base. Il numero di revisione minore corrisponde a un miglioramento incrementale dell'architettura di base, eventualmente includendo nuove funzionalità. **N.B:** non confondere la capacità di calcolo con la versione CUDA, che è la versione della piattaforma software CUDA.

Ogni SM ha una memoria on-chip dei seguenti quattro tipi:

- Un insieme di *registri locali 32-bit* per processore
- Una cache di dati paralleli o *memoria condivisa* che è condivisa da tutti i core del processore scalare ed è dove risiede lo spazio di memoria condiviso
- Una *costant cache* read-only è condivisa da tutti i core del processore scalare e accelera le letture dallo spazio di memoria costante, che è un'area di sola lettura della memoria del dispositivo
- Una read-only *texture cache* che è condiviso da tutti i core del processore scalare e accelera le letture dallo spazio di memoria texture, che è una regione di sola lettura della memoria del dispositivo